Bachelorarbeit

Tobias Pfeiffer

Entwicklung eines Algorithmus zur Erzeugung von Tiefenbildern mit Hilfe einer Stereokamera

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Tobias Pfeiffer

Entwicklung eines Algorithmus zur Erzeugung von Tiefenbildern mit Hilfe einer Stereokamera

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr.-Ing. Andreas Meisel
Zweitgutachter : Prof. Dr. rer. nat. Reinhard Baran

Abgegeben am 30. August 2007

Tobias Pfeiffer

Thema der Bachelorarbeit

Entwicklung eines Algorithmus zur Erzeugung von Tiefenbildern mit Hilfe einer Stereokamera

Stichworte

Tiefenbilder mit Hilfe der LTI-Lib und der Point Grey Research Bumblebee – Kamera erzeugen

Kurzzusammenfassung

Diese Arbeit beschreibt die Vorgehensweise, wie mit Hilfe der Bumblebee - Kamera und der LTI-Lib Tiefenbilder erzeugt werden können.

Ende des Textes

Tobias Pfeiffer

Title of the paper

Development of an algorithm to design depth images with a stereo camera

Keywords

Construction of depth images with LTI-Lib and Bumblebee camera.

Abstract

This report describes how to construct depth images with a bumblebee stereo camera in aggregation with the LTI-Lib.

End of text

Danksagung

Hiermit möchte ich allen meinen Dank aussprechen, die direkt und indirekt am Entstehen dieser Arbeit mitgewirkt haben.

Ich danke besonders meinen Eltern, die durch ihre finanzielle und materielle Unterstützung überhaupt dieses Studium und letztendlich auch diese Arbeit ermöglicht haben.

Mein Dank gilt auch meiner Verlobten Veronika-Dorothea und unserem kleinen Jan Luca, die mir ebenfalls für den jetzigen Studienabschluss ihre vollsten Kräfte und Unterstützung gegeben haben.

Mein Dank gilt auch allen Mitgliedern der Friedenskirche Buchholz und der Freien Evangelischen Gemeinde Hittfeld, die mir durch Ihre Gebete Kraft und ein Gutes Gelingen ermöglicht haben.

Während der Bearbeitungszeit dieser Arbeit ist mir ein Bibelspruch besonders zu Herzen gegangen, den ich nun zitieren möchte:

„Schreib alles auf, was du gesehen hast, was jetzt geschieht und was in Zukunft geschehen wird.“
Offenbarung 1, Vers 19 aus der Bibelübersetzung „Hoffnung für Alle“

Inhaltsverzeichnis

Kapitel 1

Aufgabenstellung

Ziel dieser Arbeit ist die Implementation und Entwicklung eines Algorithmus zur Erzeugung von Tiefenbildern unter Verwendung einer Stereokamera. Das Tiefenbild soll aus den Stereo-Bildpaaren erstellt werden, die die Stereokamera liefert, welche dann mit dem Programm Impresario[1] der RWTH Aachen weiterverarbeitet werden.

Die Erzeugung der Bilder und Weiterverarbeitung soll in zwei getrennten Modulen erfolgen.

Das erste Modul soll die Aufgabe haben, die Bilder der Kamera zu „grabben“ (Holen der Bilder von der Kamera) und anschließend den folgenden Modulen zur Verfügung zu stellen.

Das zweite Modul dieser Arbeit soll die Aufgabe haben, die vom ersten Modul zur Verfügung gestellten Bilder entgegen zu nehmen und daraus das geforderte Tiefenbild zu errechnen.

Auf die verwendete Hard- und Software wird im übernächsten Kapitel eingehend eingegangen.

In den darauf folgenden Kapiteln geht es um die verschiedensten Tiefenbildalgorithmen sowie die Umsetzung in die Praxis. Außerdem wird anschließend ein Blick in die Zukunft gewährt und ein Fazit gezogen.

[1] Impresario entwickelt an der RWTH Aachen
http://www.techinfo.rwth-aachen.de/Software/Impresario/

Kapitel 2

Einleitung

2.1 Stereosehen in der Natur

Der Vorgang des Sehens und vor allem der Tiefenbildwahrnehmung ist in der Natur ein recht komplexes Instrument, welches in der Evolution eine sehr große Bedeutung gespielt hat.

Um überhaupt Tiefen in einem Bild zu erkennen benötigt der Mensch, wie auch das Tier, mindestens zwei Augen, um eine Wahrnehmung einer Tiefe in einem Bild zu erhalten. Wie in Abbildung 2.1 [maya] dargestellt, erstellt der Mensch einen Tiefeneindruck in seinen Bildern durch die horizontale Verschiebung (hier Winkel a) und durch die Basislinie (hier c) zwischen dem linken und dem rechten Auge. d ist dann die entsprechende Entfernung.

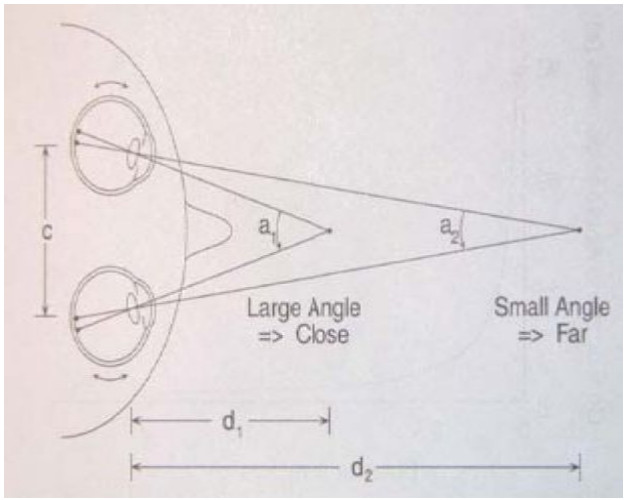


Abbildung 2.1 Darstellung des menschlichen Tiefensehens [maya]

Es ist aber auch möglich, die Tiefe aus Sicht der Netzhaut zu errechnen. Abbildung 2.2 [maya] zeigt die unterschiedlichen Abstände der Gegenstände auf der Netzhaut je nach Entfernung des Gegenstandes von den Augen.

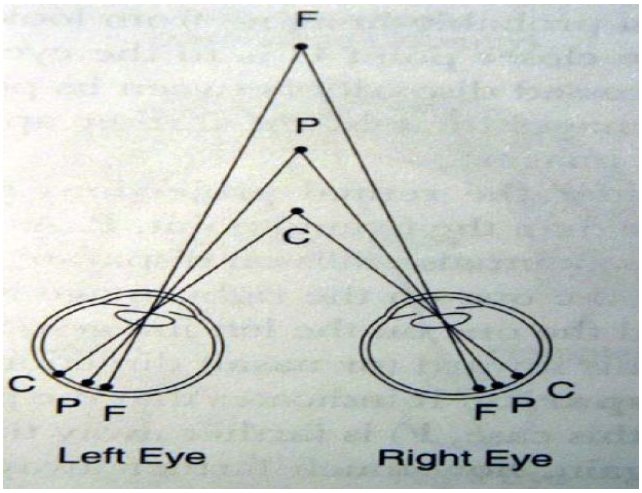


Abbildung 2.2 Tiefe aus Sicht der Netzhaut [maya]

Aus Abbildung 2.2 wird deutlich, dass je näher ein Gegenstand dem Auge ist, desto weiter ist der korrespondierende Bildpunkt von der Augenmitte entfernt auf der Netzhaut abgebildet. Punkt C befindet sich nahe am Auge. Aus diesem Grunde sind die abgebildeten Punkte auf der Netzhaut weit von der Augenmitte entfernt. Punkt F liegt sehr weit vom Auge entfernt. Aus diesem Grunde sind die abgebildeten Punkte auf der Netzhaut in der Nähe der Augenmitte.

2.2 Prinzip der Stereofotografie

Bereits Ende des 19. Jahrhunderts befassten sich erste Menschen mit dem Prinzip der Stereofotografie.

Bei der Stereofotografie werden zwei Kameralinsen mit einem festgelegten Abstand voneinander in ein Kameragehäuse verbaut. Die Bilder werden auf normalem Fotofilm festgehalten.

Abbildung 2.3 [wiki_stereo] zeigt eine analoge Stereokamera, wie sie im 20. Jahrhundert verwendet wurde.

Abbildung 2.3 Darstellung einer analogen Stereokamera [wiki_stereo]

Bereits zu Anfang der Stereofotografie entstanden erste Bilder mit den entsprechenden Kameras.

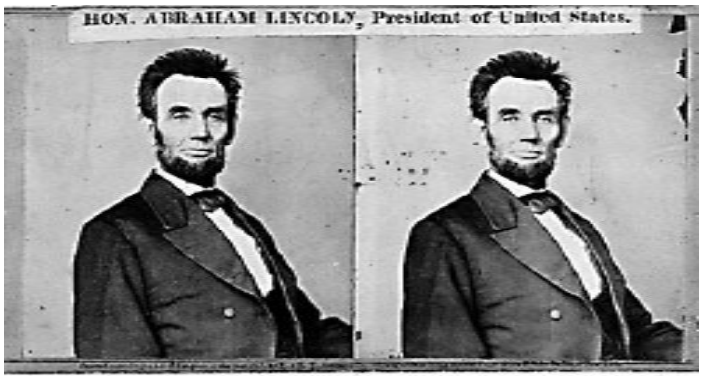


Abbildung 2.4 Präsident Abraham Lincoln als Stereofotografie [maya]

Abbildung 2.4 [maya] zeigt wahrscheinlich eines der ersten Bilder, die jemals in der Geschichte der Menschheit mit Hilfe der Stereofotografie aufgenommen wurden.

In Zeiten der modernen Datenverarbeitung gibt es mittlerweile auch entsprechende Stereokameras in digitalen Versionen.

Einer der bekannten Hersteller von solchen Kameras ist die kanadische Firma Point Grey Research[1]. Diese Firma stellt auch Kameras mit drei Linsen her, die eine spätere Verarbeitung der Bilder in Tiefenbilder noch vereinfachen können, in dieser Arbeit aber nicht zur Verwendung kommen.

Abbildung 2.5 [ptgrey] zeigt eine Firewire-Stereokamera mit 2 Linsen, die so genannte Bumblebee. Diese liegt auch dieser Arbeit zu Grunde.

Abbildung 2.5 Bumblebee Kamera [ptgrey]

[1] Point Grey Research, Vancouver, Kanada, http://www.ptgrey.com

Abbildung 2.6 [ptgrey] zeigt eine Firewire-Stereokamera mit 3 Linsen, um eine noch bessere Tiefenbildauflösung zu erhalten. Diese wird jedoch in dieser Arbeit nicht verwendet.

Abbildung 2.6 Bumblebee Kamera mit 3 Linsen [ptgrey]

2.3 Tiefenbilder

Wenn mit einer Stereokamera Bilder aufgenommen werden und diese übereinander gelegt werden, wird einem schnell bewusst, dass die Bilder gar nicht übereinander passen. Selbst wenn die Ränder exakt übereinander liegen ist, je nachdem wie der entsprechende Gegenstand entfernt ist, ein Versatz, auch Disparität genannt, in den Bildern auf horizontaler Ebene zu bemerken.

Dieser Versatz kommt dadurch zustande, dass die aufnehmenden Kameras nicht direkt ineinander verschachtelt verbaut sind, sondern mit einem Abstand, der Basislinie, nebeneinander angeordnet sind. Dieser Versatz bestimmt letztendlich wie weit ein Objekt von der Kamera entfernt ist.

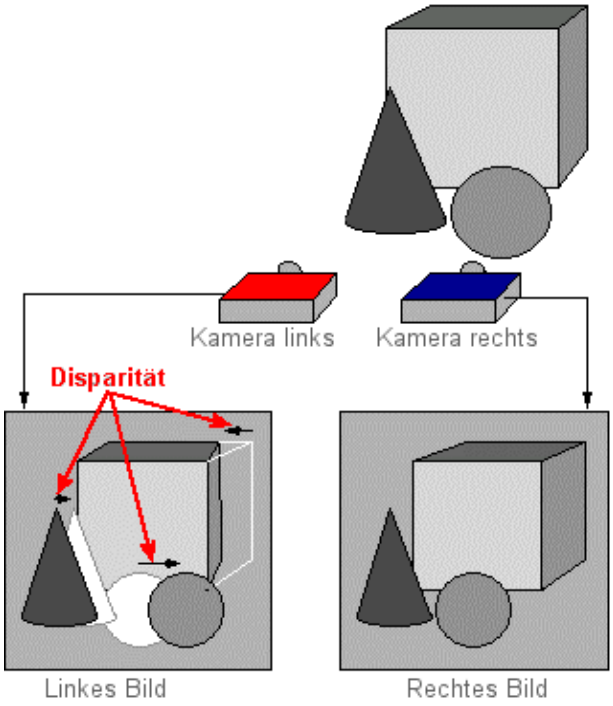


Abbildung 2.7 Disparität im linken Bild mit schwarzen Pfeilen dargestellt [predrag]

Abbildung 2.7 [predrag] zeigt im linken Bild die Unterschiede zum rechten Bild in Form von hell markierten Flächen.

Am folgenden Beispiel erkennt man die Disparität auch in einem realen Bild:

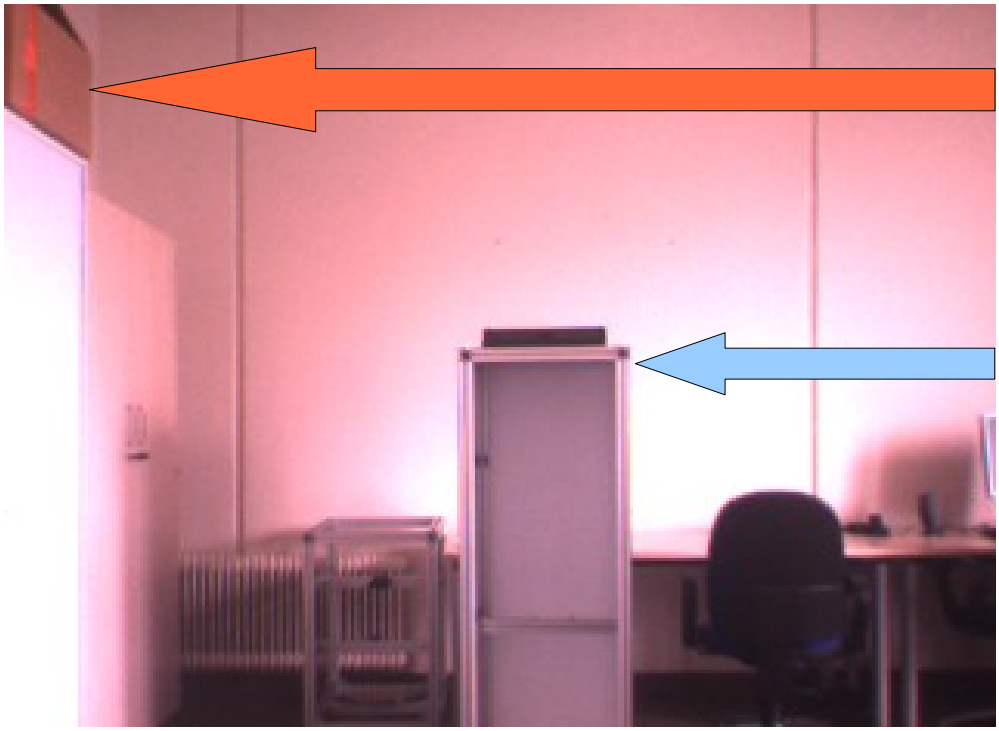


Abbildung 2.8 linkes Kamerabild [pfeiffer]

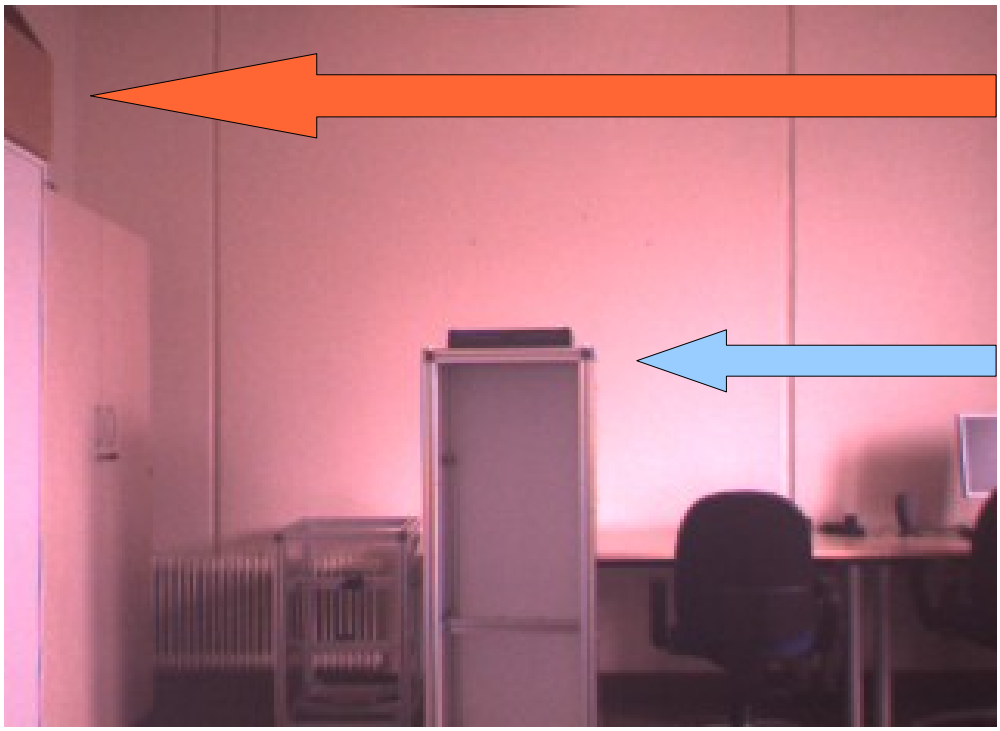


Abbildung 2.9 rechtes Kamerabild [pfeiffer]

Die beiden Pfeile in den Bildern machen die Disparität deutlich. Sie sind in beiden Bildern gleich lang, zeigen aber in den Bildern nicht auf dieselben Punkte.

Die Tiefe in dem Bild lässt sich leicht über folgende Formel [predrag] bestimmen:

$$Tiefe = \frac{(Basislinie * Brennweite)}{Dispariät}$$

Diese Formel wurde aus der folgenden Formel [predrag] hergeleitet:

$$Disparität = \frac{(Basislinie * Brennweite)}{Tiefe}$$

Die Basislinie ist der Abstand der beiden Kameras voneinander (bei der Bumblebee 12 cm). Die Brennweite ist die Brennweite der verwendeten Kamera (bei der Bumblebee 2,0 mm bei 100% Auflösung, 4,0 mm bei 70% Auflösung oder 6,0 mm bei 50% Auflösung). Die Disparität ist der Abstand in Anzahl Pixel zwischen dem linken und dem rechten Bild. Aus diesen Daten und mit den entsprechenden Formeln kann dann ein Tiefenbild erstellt werden, dass in etwa wie folgt aussehen könnte:

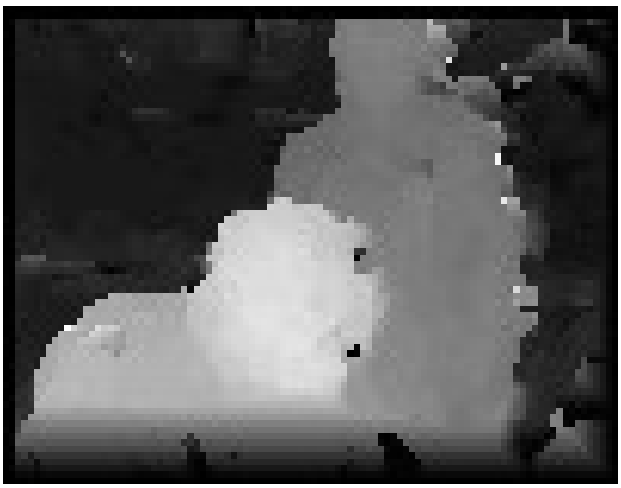


Abbildung 2.10 Tiefenbild [ptgrey]

Aus diesem Tiefenbild (Abbildung 2.10) ist zu erkennen, wie weit die entsprechenden Gegenstände von der Kamera entfernt sind. In diesem Beispielbild sind die Entfernungen durch unterschiedliche Graustufen dargestellt. Gegenstände, die näher an der Kamera liegen, sind heller dargestellt als Gegenstände, die weiter von der Kamera entfernt liegen.

Kapitel 3

Verwendete Hard- und Software

3.1 Verwendete Hardware

3.1.1 Verwendeter Computer

Hersteller	HP Compaq
CPU	Intel Pentium 4 2,8 GHz
Arbeitsspeicher	1 GB
Betriebssystem	Microsoft Windows XP SP2

3.1.2 Verwendete Kamera

Die in dieser Arbeit benutzte Kamera stammt von der Firma Point Grey Research aus Kanada. Die Kamera trägt die Bezeichnung Bumblebee, welche in Abbildung 3.1 [ptgrey] dargestellt ist.

Abbildung 3.1 Bumblebee Kamera [ptgrey]

Die technischen Daten dieser Kamera sind:

Hersteller	Point Grey Research, Kanada
Anschlussart	IEEE 1394 (Firewire)
Objektive	2x Sony ICX084 Farb- oder Schwarzweiß-CCDs 640x480 Pixel
Objektivabstand	12 cm
Abmessungen	160 x 40 x 50 mm
Gewicht	375 g

Des weiteren verfügt die Kamera über folgende interessante Merkmale:

- Individuelle Kontrolle der einzelnen CCDs zur individuellen Helligkeitssteuerung
- Automatische / manuelle Blendenverschlusssteuerung mit einstellbarer Bildwiderholungsrate

3.2 Verwendete Software

3.2.1 Microsoft Visual Studio .NET 2003

Als Entwicklungsumgebung der in dieser Arbeit verwendeten Module für Impresario dient das Programm Visual Studio .NET 2003 der Firma Microsoft.

Verwendete Programmiersprache ist C++.

Die Entwicklungsumgebung wird verwendet um den Modulen die Funktionalität zu geben. Außerdem werden durch die Entwicklungsumgebung sogenannte Dynamic Linked Libraries (kurz DLL) erstellt, welche von Impresario benötigt werden, um überhaupt die Module laden zu können.

3.2.2 Impresario

Impresario ist das Hauptprogramm, welches dieser Arbeit zugrunde liegt. Es wurde von der RWTHH Aachen entwickelt und ist unter folgender URL herunterzuladen: http://www.techinfo.rwth-aachen.de/Software/Impresario/. Impresario ist eine grafische Benutzeroberfläche für die LTI-Lib[1], welche ebenfalls an der RWTH Aachen entwickelt wurde. Impresario ist ein Bildverarbeitungsprogramm, welches durch selbst geschriebene Module erweitert werden kann. Selbst geschriebene Module können z.B. sein:

[1] LTI-Lib, http://ltilib.sourceforge.net/doc/homepage/index.shtml

- Module zum Grabben von Bildern einer Kamera
- Module zum Filtern von bestimmten Bildmerkmalen aus Bildern
- Module zum Weiterverarbeiten von Bildern, zum Beispiel wie es in der vorliegenden Arbeit geschieht, der Berechnung von Tiefenbildern
- Module zum Abspeichern von Bildern

Impresario besticht durch seine einfache Benutzeroberfläche. Module können einfach per Anklicken, Herüberziehen und Ablegen (Drag and Drop) in den aktuellen Ablauf integriert werden. Die Aus- und Eingänge dieser Module können auf ebenso einfache Weise miteinander verbunden werden.

Da Impresario unter Microsoft Visual Studio .NET 2003 entwickelt wurde, ist für die Entwicklung eigener Module diese Entwicklungsumgebung zwingend erforderlich, damit die selbst entwickelten Module auch reibungslos laufen können. Um unter Impresario eigene Module entwickeln zu können, wird zudem eine Version der Programmiersprache Perl auf dem Entwicklungscomputer voraus gesetzt. Für diese Arbeit wurde ActivePerl 5.8.8 Build 820, zu beziehen unter http://downloads.activestate.com/ActivePerl/Windows/5.8/, verwendet.

Impresario hat folgendes Aussehen auf Windows-Computern:

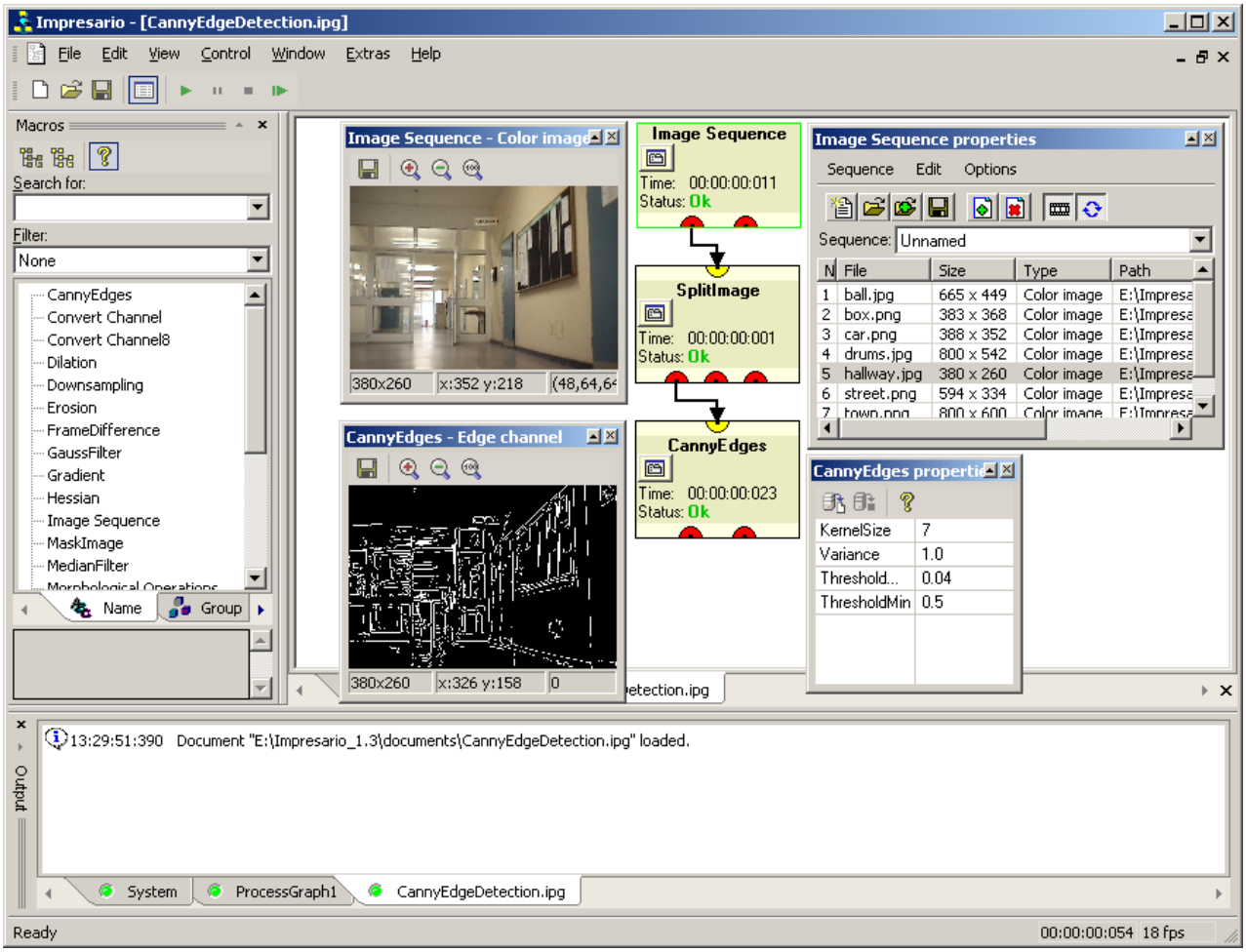


Abbildung 3.2 Impresario [impresario]

Wie in Abbildung 3.2 [impresario] zu sehen ist, werden folgende Schritte nacheinander abgearbeitet:

1. Mit „Image Sequence“ werden unterschiedliche Bilder aus einer Liste sequentiell geladen und für die weitere Verarbeitung zur Verfügung gestellt.
2. In „SplitImage“ wird das eingegebene RGB-Bild in seine drei Einzelfarbteile (Rot, Grün und Blau) aufgeteilt.
3. „CannyEdges“ ist ein Bildfilter, der in einem eingegebenen Bild Kanten erkennt. Vorhandene Kanten werden in dem daraus resultierenden Bild Weiß dargestellt. Der Rest des Bildes wird Schwarz eingefärbt.

3.2.3 LTI-Lib

Die LTI-Lib [ltilib] wurde ebenfalls von der RWTH Aachen entwickelt. Die LTI-Lib ist eine Bibliothek für C++, welche eine große Anzahl von Funktionen bereit stellt, die für eine sinnvolle Bildverarbeitung gut sind. Hierzu zählen unter anderem Funktionen zum Laden und Speichern von Bildern, Filterfunktionen, wie z.B. verschiedene Kantendetektoren, Funktionen zum Grabben von Bildern von USB-Kameras und noch vieles mehr.

Für diese Arbeit wird die aktuelle öffentliche Version 1.9 verwendet. Die neuere Version LTI-Lib 2 ist leider noch nicht öffentlich verfügbar und noch im Entwicklungsstadium. Sie steht leider nur Angehörigen der RWTH Aachen zur Verfügung.

Für die Zukunft wird sich aber sicher die neuere Version, sobald sie freigegeben ist, durchsetzen, weil sie mit mehr und auch verbesserten Funktionsumfang aufwarten kann. Hierzu zählt unter anderem der erste Versuch der Verbindung und des Zugriffs auf Firewire-Kameras, aber auch noch mehr unterstützte Bildformate.

3.2.4 Digiclops Stereo Vision SDK und Triclops Stereo Vision SDK

SDK bedeutet Software Development Kit.

Das Digiclops Stereo Vision SDK (nachfolgend Digiclops genannt) sowie das Triclops Stereo Vision SDK (nachfolgend Triclops genannt) sind Entwicklungsbibliotheken, welche der Bumblebee-Kamera beiliegen. Sie dienen dazu,.die Produkte der Firma Point Grey Research unter der Programmiersprache C++ anzusprechen und zu steuern. Zudem bieten Sie auch etliche nützliche Routinen, welche in der Bildverarbeitung sinnvoll sind, wie z.B. dem so genannten „Rektifizieren“ von Bildern.

„Rektifizieren“ bedeutet, dass die Bilder einer Stereokamera auf vertikaler Ebene einheitlich ausgerichtet werden und dadurch nur noch ein Versatz auf horizontaler Ebene vorhanden ist, welcher für die Tiefenbildberechnung erforderlich ist.

Digiclops ist eine Bibliothek, die die Grundfunktionalitäten der Kameras von Point Grey Research unter der Programmiersprache C++ zur Verfügung stellt. Hierzu zählen das Einbinden der Kameras, das Justieren der Kameras, das Speichern von aufgenommenen Bildern sowie das Grabben von Bildern. Funktionen dieser Bibliothek sind zum Beispiel:

- digiclopsInitialize (DigiclopsContext context, unsigned long ulDevice),

welche die am Bus vorhandene Kamera initialisiert und verbindet.
- digiclopsStart (DigiclopsContext context), welche das Grabben der Bilder startet.
- digiclopsGrabImage (DigiclopsContext context), welche die aktuellsten Bilder der angeschlossenen Kamera grabbt.

Triclops ist eine Bibliothek, die die Digiclops um Funktionen der Bildverarbeitung ergänzt. Sie ersetzt allerdings nicht Digiclops sondern erweitert deren Funktionsumfang. Funktionen dieser Bibliothek sind zum Beispiel:

- triclopsStereo (TriclopsContext context), welche nach vorher eingestellten Parametern Tiefenbilder erzeugt. Zu den Aufgaben dieser Funktion zählen: Tiefenbildverarbeitung, Validieren und Subpixelinterpolation.
- triclopsRectifyColorImage (TriclopsContext context, TriclopsCamera nCamera, TriclopsInput* input, TriclopsColorImage* output), welche eingegebene Farbbilder rektifiziert.
- triclopsGetImage(const TriclopsContext context, TriclopsImageType imageType, TriclopsCamera camera, TriclopsImage* image), welche ein in den Parametern definiertes Bild aus dem Kontext holt und in ein separates Bild speichert.

3.2.5 Active Perl 5.8.8 Build 820

Das Programm Active Perl stellt auf dem Entwicklungscomputer die Programmiersprache Perl zur Verfügung. Diese wird benötigt, um eigene Projekte unter Impresario erstellen zu können. Es kann herunter geladen werden unter: http://downloads.activestate.com/ActivePerl/Windows/5.8/.

Kapitel 4

Algorithmen zur Berechnung der Disparität

4.1 Algorithmus nach M. Pilon und P. Cohen

Der Algorithmus von M. Pilon und P. Cohen wurde bereits 1996 veröffentlicht und ist in der Arbeit von Michael Kux [kux] beschrieben. Laut Michael Kux arbeitet der Algorithmus mit zwei Formeln, die laut Kux auf damaligen Rechnern langsam waren, aber laut Kux bei heutigen Rechnern sehr schnell laufen sollen. Die Formeln lauten:

$$x*(\alpha_1 x_1+\alpha_2 y_1+\alpha_3)+x*x_2*(\alpha_4 x_1+\alpha_5 y_1+\alpha_6)+x_2*(\alpha_7 x_1+\alpha_8 y_1+\alpha_9)+\alpha_{10} x_1+\alpha_{11} y_1+\alpha_{12}=0$$

$$y*(\beta_1 x_1+\beta_2 y_1+\beta_3)+y*x_2*(\beta_4 x_1+\beta_5 y_1+\beta_6)+x_2*(\beta_7 x_1+\beta_8 y_1+\beta_9)+\beta_{10} x_1+\beta_{11} y_1+\beta_{12}=0$$

Die einzelnen Bestandteile haben folgende Bedeutung:

x_1 und y_1 sind Elemente von Bild 1. x_2 und y_2 sind die korrespondierenden Punkte in Bild 2. x und y sind die entsprechenden Punkte im Tiefenbild. Die restlichen α_n und β_n sind Konstanten, die in jedem Punkt des Bildes gleich sind. Wenn der Algorithmus gestartet wird, sollen laut Kux nach dem ersten Berechnungsschritt 7 der 24 Konstanten bekannt sein. Die anderen 17 unbekannten Konstanten müssten mit Hilfe eines linearen Gleichungssystems ermittelt werden.

Da allein schon der Weg, um diesen Algorithmus zu entschlüsseln sehr kompliziert ist, wurde dieser Algorithmus in dieser Arbeit nicht implementiert. Die Aussagen über diesen Algorithmus, sind die Resultate, welche Michael Kux in seiner Arbeit getroffen hat.

4.2 Sum of Absolute Differences

Dieses ist ein sehr schöner und auch effektiver Algorithmus zur Berechnung von Disparitäten. Er arbeitet nach einer einfachen Formel, welche bei [triclops] beschrieben wird. Diese lautet:

$$\min_{d=d_{min}}^{d_{max}} \sum_{i=\frac{-m}{2}}^{\frac{m}{2}} \sum_{j=\frac{-m}{2}}^{\frac{m}{2}} \left| I_r[x+i][y+j]-I_l[x+i+d][y+j] \right|$$

d_{min} und d_{max} sind die minimale und die maximale Disparität. m ist die Maskengröße. I_r und I_l sind mit den Koordinaten in den eckigen Klammern die entsprechenden Bildpunkte.

Bei diesem Algorithmus müssen die eingegebenen Bilder rektifiziert sein.

Es wurde versucht diesen Algorithmus für die hier vorliegende Arbeit umzusetzen, allerdings ist dieser bei ziemlich homogenen Bildern zu präzise und findet nur wenige Stellen, wo er die Disparität errechnen kann. Zudem ist der Algorithmus nicht besonders schnell. Bei der Umsetzung in Impresario brauchte er bei einem Tiefenbild bis zu 3 Minuten.

4.3 Algorithmus nach Alexandre Bernardino und José Santos-Victor

Der Algorithmus dieser beiden Herren [bernardino] setzt auf rektifizierte Ausgangsbilder auf. In diesem Algorithmus steht I für das linke und I' für das rechte Ausgangsbild.

Die Disparität wird hier wie folgt formuliert: $d(x)=(d_x, d_y)$.

Wenn der Farbwert in den jeweiligen Bildern an einer bestimmten Position als **x** beziehungsweise **x'** bezeichnet wird, so ergibt sich für die Disparität folgende Formel: $d(x)=x'-x$

x und **x'** sind hierbei die entsprechenden Punkte in beiden Bilder, auf denen dieselbe Ecke beziehungsweise derselbe Bildpunkt dargestellt ist. Wenn ein Bildpunkt im Referenzbild nicht dargestellt wird, so gilt, dass die Disparität in diesem Punkt nicht definiert ist und der entsprechende Bildpunkt verdeckt ist. Hierbei gilt folgende Formel: $d(x)=0$.

4.4 Disparity Space Images (Verschiebungs-Raum-Bilder) [thiel]

Ein Disparity Space Image (DSI) ist ein Bild, welches die Übereinstimmungen von Gegenständen und Pixeln in beiden Bildern darstellt. Es benutzt ebenfalls rektifizierte Bilder als Grundlage. DSI funktioniert nach folgendem Prinzip. Hierbei sind s^L_i und s^R_i die jeweiligen Punkte im linken und rechten Bild.

- Man nehme in einem Bild einen Punkt und schiebe ihn pixelweise auf den entsprechenden Punkt im anderen Bild.
- In jedem dieser Schritte wird die Differenz der beiden Punkte in der nächsten Zeile des DSI gespeichert.
- Es entsteht somit ein Übereinstimmungsraum.

Als eine Formel formuliert sieht dieses wie folgt aus:

$$DSI_i^R(x, d) = I_R(x, i) - I_L(x+d, i)$$

für 0<=(x+d)<N, 0<=d<D und 0<=x<N. Das hochgestellte R in DSI^R zeigt an, dass es sich um das rechte DSI handelt. Das DSI^L ist einfach die verschobene und gesicherte Version des DSI^R.

Hieraus kann auch eine Disparität errechnet werden.

4.5 Algorithmus nach Michael Bleyer und Marigrit Gelautz [bleyer]

Der Algorithmus dieser Arbeit arbeitet nach folgender Struktur:

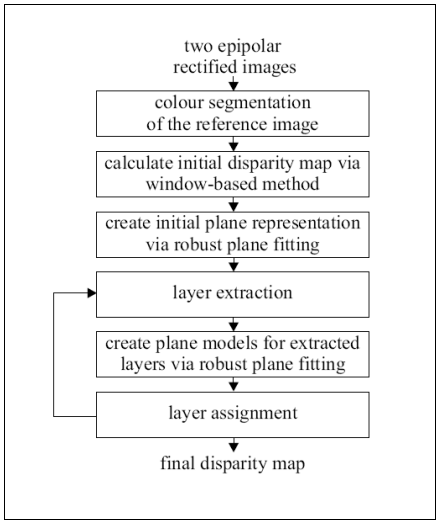


Abbildung 4.1 Algorithmus nach Bleyer [bleyer]

Abbildung 6.1 [bleyer] zeigt den Ablauf des Algorithmus von Michael Bleyer und Margrit Gelautz. Ihn genauer zu beschreiben, würde den Rahmen dieser Arbeit sprengen. Zudem ist das Diagramm eindeutig und klar strukturiert.

4.6 Algorithmus nach Tobias Pfeiffer

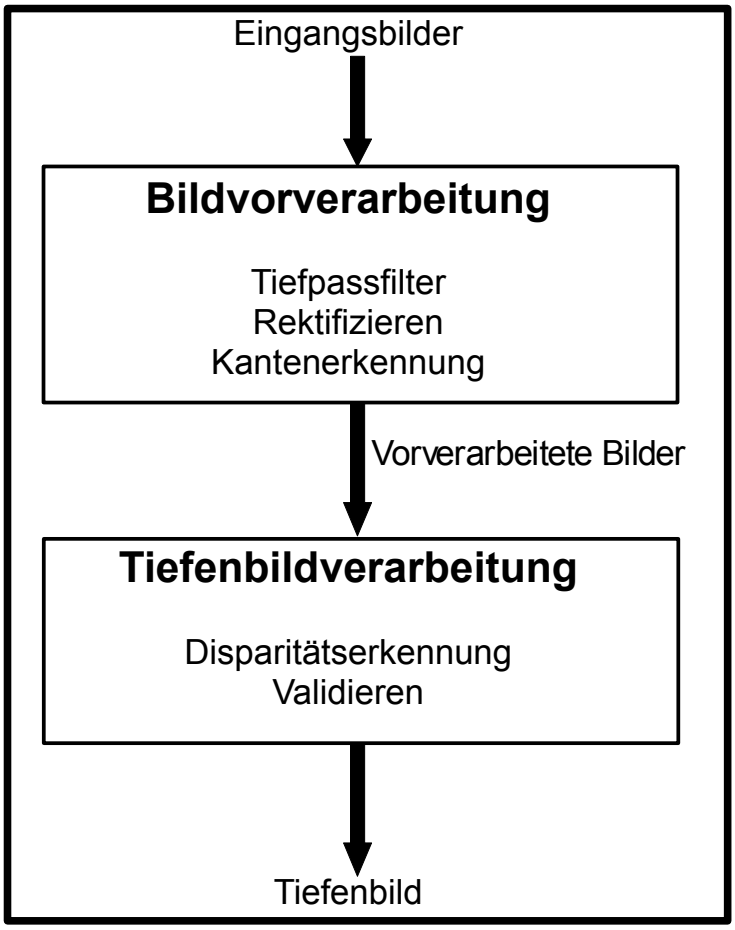
Einer der in dieser Arbeit umgesetzten Algorithmen ist der Versuch, selbst einen Tiefenbildalgorithmus zu entwickeln. Der Algorithmus benötigt als Grundlage zwei rektifizierte Bilder, eines von der linken und eines von der rechten Kamera. Diese Bilder werden dann von dem Algorithmus weiterverarbeitet. Um diesen Algorithmus zu verstehen, muss die Vorgehensweise erklärt werden. Der Algorithmus verfährt nach folgendem Verfahren:

1. Die rektifizierten Quellbilder werden entgegen genommen.
2. In den beiden Bildern werden mit Hilfe von Kantenfiltern für jedes Bild einzeln die Kanten erkannt.
3. Nun werden in jedem Bild die Kanten miteinander verglichen. Bei gleichartigen Kanten wird sich die Positionsdifferenz der Kanten in den beiden Bildern für jeden Bildpunkt im Bild gemerkt.
4. Die nun gewonnenen Differenzen, auch Disparitäten genannt, dienen als Grundlage zur Tiefenbildberechnung.

5. Mit der Formel $Tiefe = \frac{Basislinie * Brennweite}{Disparität}$ werden nun die Tiefenbildinformationen berechnet und in das Zielbild übertragen.
6. Das nun fertige Zielbild wird am Bildschirm angezeigt.

4.7 Algorithmus nach Point Grey Research [triclops]

Dieser Algorithmus wurde ebenfalls in dieser Arbeit implementiert. Der Algorithmus geht nach folgendem Verfahren vor:



Die einzelnen Punkte dieser Vorgehensweise werden im folgenden erklärt:

Bildvorverarbeitung:

Dieses Modul bereitet die Bilder für die Tiefenbildverarbeitung vor. Das Modul erlaubt Spezifikationen der Verarbeitungsauflösung und die Funktionalität des Tiefpassfilters. Um die Bilder rektifizieren zu können, müssen sie in den Helligkeiten vereinheitlicht werden. Hierzu ist der Tiefpassfilter sinnvoll, er muss aber nicht verwendet werden.

Beim Rektifizieren geht es um die vertikale Ausrichtung beider Bilder. Zudem werden gerade Linien in der Realität meistens in den Eingangsbildern nicht ganz gerade sondern als Bogen dargestellt. Diese Fehldarstellung wird auch behoben.

Bei der Kantenerkennung, welche Optional ist, geht es um die Kantenerkennung in beiden Bildern. Die Kantenerkennung wird erforderlich, wenn sich die Hinter-

grundhelligkeit beständig ändert, so dass die automatische Helligkeitssteuerung der Bumblebeekamera dies nicht korrekt ausgleichen kann. Hier ist die Erkennung von Kanten von Vorteil, da hiermit, trotz der schlechten Helligkeitsunterschiede, eine Tiefenbildverarbeitung durchaus gewährleistet wird.

Tiefenbildverarbeitung:

In diesem Modul wird das eigentliche Tiefenbild berechnet. Bei der Disparitäts-erkennung wird wie folgt vorgegangen:

1. Suche für jeden Pixel mit seinen benachbarten Pixeln im Referenzbild im zweiten Bild in derselben Reihe ein ähnliches Pixel mit benachbarten Pix-eln.
2. Wähle das bestmögliche Pixel aus.
3. Wiederhole Schritt 1 und 2 bis zum Ende des Bildes.
4. Ende.

Als Formel ist dies die so genannte „Sum of Absolute Differences“, die wie folgt aussieht: $\min_{d=d_{min}}^{d_{max}} \sum_{i=-\frac{m}{2}}^{\frac{m}{2}} \sum_{j=-\frac{m}{2}}^{\frac{m}{2}} \left| I_r[x+i][y+j] - I_l[x+i+d][y+j] \right|$, wobei hier d_{min} und d_{max} die minimale beziehungsweise maximale Disparität sind. Die Größe der Maske wird durch m dargestellt. I_r und I_l sind hierbei das rechte beziehungsweise linke Bild.

Der Schritt des Validierens ist in einigen Fällen notwendig, wenn Bildbereiche in einem Bild vorhanden, in dem anderen Bild aber durch Gegenstände verdeckt sind. Beim Texturvalidieren wird nachgesehen, ob Disparitäten gültig sind. Bei Unstimmigkeiten werden Disparitäten, die nicht gültig sind ungültig gemacht. Hier-bei wird auf die Ergebnisse der „Sum of Absolute Differences“ zurück gegriffen.

Kapitel 5

Grabben der Kamerabilder

5.1 Der Algorithmus

Zunächst wird erst einmal der Digiclops-Container erstellt, in dem sämtliche kameraspezifische Daten gespeichert werden. Hier wird unter anderem die Kamera initialisiert und gestartet. Es wird festgelegt, welche der Kamerabilder verwendet werden sollen, welche Auflösung die Bilder haben sollen und welche Bildwiederholungsrate verwendet werden soll. Im nächsten Schritt werden die Bilder gegrabbt und in Triclops-Container gespeichert. Die Triclops-Container sind im Prinzip identisch mit den Digiclops-Containern. Nun werden die gewonnenen Bilder rektifiziert und auf die Ausgangskänale des Moduls von Impresario gelegt.

5.2 Implementation als Sourcecode

```
bool CKamera::InitMacro()
{
	percentRate = 0.0;
	e  = digiclopsCreateContext( &digiclops );  // Generieren des Digiclops-Containers
	// initialize Digiclops device 0
	e  = digiclopsInitialize( digiclops, 0 );
	// start the Digiclops device streaming images
	e  = digiclopsStart( digiclops );
	// tell the Digiclops to provide all image types
	e  = digiclopsSetImageTypes( digiclops, ALL_IMAGES );
	e = digiclopsSetImageResolution( digiclops, DIGICLOPS_FULL );  // Einstellung der Auflösung
	// set the frame rate if asked
	DigiclopsFrameRate frameRate;
	if( percentRate > 0.0 ) {
		if( percentRate <= 12.0 ) {frameRate = DIGICLOPS_FRAMERATE_012;}
		else if( percentRate <= 25.0 ) {frameRate = DIGICLOPS_FRAMERATE_025;}
		else if( percentRate <= 50.0 ) {frameRate = DIGICLOPS_FRAMERATE_050; }
		else {frameRate = DIGICLOPS_FRAMERATE_100;}
		e = digiclopsSetFrameRate( digiclops, frameRate );
	}
	// retrieve the current frame rate
	e = digiclopsGetFrameRate( digiclops, &frameRate );
	switch( frameRate ) {
		case DIGICLOPS_FRAMERATE_100 :	percentRate = 100;
						break;
		case DIGICLOPS_FRAMERATE_050 :	percentRate = 50;
						break;
		case DIGICLOPS_FRAMERATE_025 :	percentRate = 25;
						break;
```

```
                default :  // case DIGICLOPS_FRAMERATE_012 :
                                                percentRate = 12;
                                                break;
        }
        printf( "Current bus rate : %lf %%\n", percentRate );
        return true;
}

bool CKamera::Apply()
{
        printf( "Grabbing an image...\n" );
        e = digiclopsGrabImage( digiclops ); // Grabben der Bilder
        e = digiclopsExtractTriclopsInput( digiclops, RIGHT_IMAGE, &triclopsInput_right ); // Extrahieren in
        e = digiclopsExtractTriclopsInput( digiclops, LEFT_IMAGE, &triclopsInput_left ); // TriclopsInput
        digiclopsGetTriclopsContextFromCamera(digiclops,&triclops); // Extrahieren Triclops-Container
        triclopsRectifyColorImage(triclops, TriCam_RIGHT, &triclopsInput_right, &tri_right); // Rektifizieren der
        triclopsRectifyColorImage(triclops, TriCam_LEFT, &triclopsInput_left, &tri_left); // Bilder
        m_channelOutput_left.resize(tri_left.nrows,tri_left.ncols); // Anpassen der Größe der Ausgangsbilder
        m_channelOutput_right.resize(tri_right.nrows,tri_right.ncols);

        int r,c; // Kopieren der Point Grey Bilder in die Ausgangsbilder
        for (r=0; r<tri_left.nrows; r++) {
                unsigned char* pucSrc_blue = tri_left.blue + r * tri_left.rowinc;
                unsigned char* pucSrc_red = tri_left.red + r * tri_left.rowinc;
                unsigned char* pucSrc_green = tri_left.green + r * tri_left.rowinc;
                for (c=0; c<tri_left.ncols; c++) {
                        m_channelOutput_left.at(r,c) = (pucSrc_blue[c]+pucSrc_red[c]+pucSrc_green[c])/3;
                }
        }
        for (r=0; r<tri_right.nrows; r++) {
                unsigned char* pucSrc_blue = tri_right.blue + r * tri_right.rowinc;
                unsigned char* pucSrc_red = tri_right.red + r * tri_right.rowinc;
                unsigned char* pucSrc_green = tri_right.green + r * tri_right.rowinc;
                for (c=0; c<tri_right.ncols; c++) {
                        m_channelOutput_right.at(r,c) = (pucSrc_blue[c]+pucSrc_red[c]+pucSrc_green[c])/3;
                }
        }
        return true;
}

bool CKamera::ExitMacro()
{
        e = digiclopsStop( digiclops ); // Beenden der Point Grey Devices
        e  = digiclopsDestroyContext( digiclops );
        return true;
}
```

Um den Code so kompakt wie möglich zu halten, wurden Fehlerbehandlungen hier nicht berücksichtigt.

5.3 Beispielbild

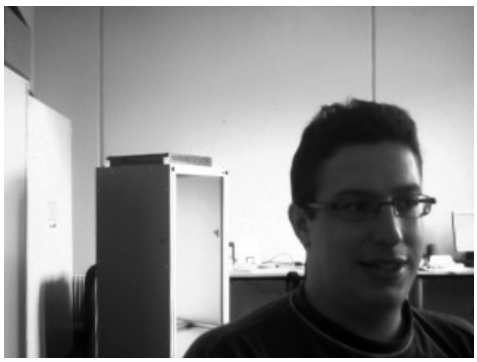


Abbildung 5.1 Linkes Kamerabild [pfeiffer]

Abbildung 5.2 Rechtes Kamerabild [pfeiffer]

5.4 Versuchsaufbau

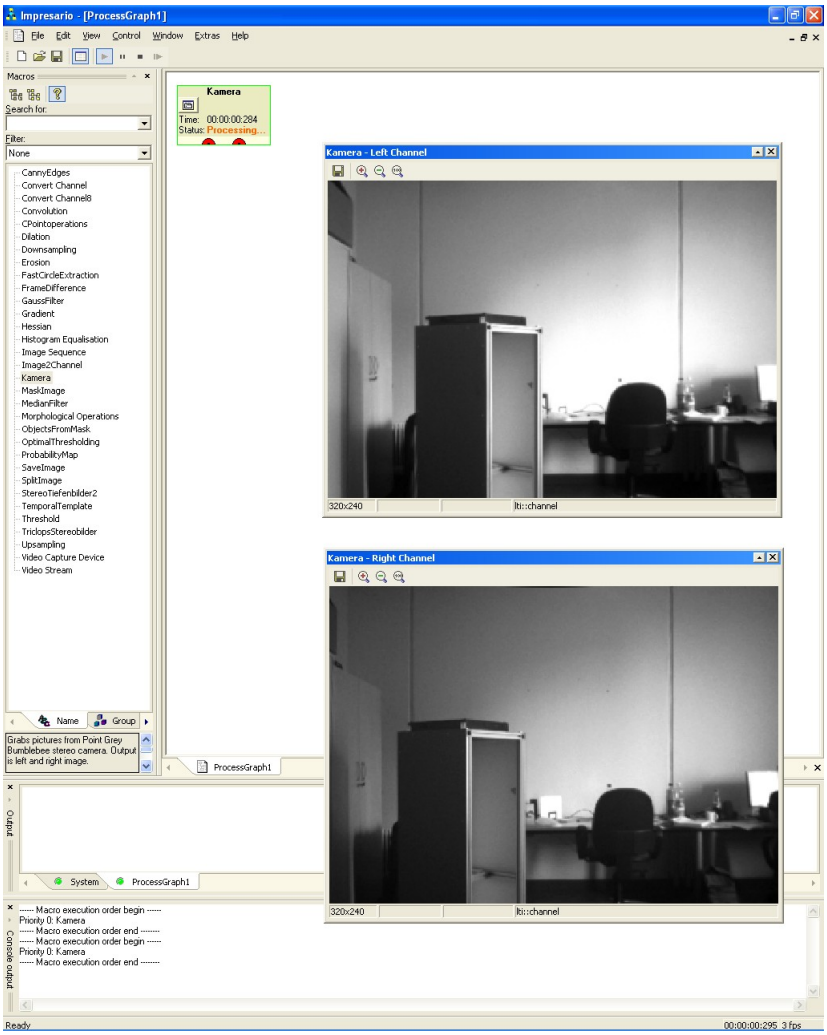


Abbildung 5.3 Versuchsaufbau Bilder Grabben [pfeiffer]

5.5 Interpretation des Ergebnisses

Das Grabben der Bilder von der Kamera funktioniert einwandfrei. Die rektifizierten Bilder werden in weniger als einer Sekunde zur Verfügung gestellt. Zwar wäre ohne das Kopieren der Bilder aus dem Point Grey eigenen Format in das Impresario eigene Format der Vorgang noch schneller, allerdings sind die Formatierungsarten beider Bildtypen zu unterschiedlich, als dass ein direktes arbeiten auf den Impresario eigenen Bildformaten möglich wäre. Aber auch mit dem Kopiervorgang ist das Grabben der Bilder schnell genug um fast Echtzeitbilder zu erhalten.

Ein kleiner Nachteil bleibt jedoch bestehen. Die Kamera liefert eine Auflösung von 640 mal 480 Bildpunkten. Die rektifizierten Bilder jedoch haben nur eine Auflösung von 320 mal 240 Bildpunkten. Hierbei gehen Bildinformationen verloren, die jedoch bei den Ergebnissen der Tiefenbildberechnung nicht störend sein sollten.

5.6 Hinweise zur Lösung

Damit das Programm übersetzt und das Modul zum Laufen gebracht werden kann, ist es zwingend erforderlich, dass die Datei „strmbasd.lib“ in das Visual Studio Projekt mit eingebunden ist. Diese Datei wird jedoch nirgends zur Verfügung gestellt. Die Datei muss selbst erzeugt werden, indem die Baseclasses des Microsoft Direct X SDK compiliert werden.

Kapitel 6

Algorithmus nach Tobias Pfeiffer

6.1 Beschreibung des Algorithmus

Der von mir implementierte Algorithmus hat folgenden Aufbau:

1. Casten der Eingangsbilder auf Bilder, die während der Laufzeit bearbeitet werden können.
2. Anpassen der Größe vom Ausgangsbild, Kantenbild und Disparitätenbild an die Größe der Eingangsbilder.
3. Setzen von jedem Bildpunkt im Ausgangsbild auf Weiß und im Disparitätenbild auf Schwarz.
4. Wende folgende Kantenfilter auf die Ausgangsbilder an:
    - Susan Edges
    - Classic Edge Detector
    - Canny Edges
5. Gehe im rechten Kantenbild von links nach rechts und von oben nach unten und finde eine Kante.
6. Wenn Kante gefunden, gehe im linken Bild in derselben Reihe nach rechts und finde ebenfalls diese Kante.
7. Merke die Verschiebung im Disparitätenbild.
8. Wiederhole Schritt 5 bis 7 bis zum Ende des Bildes.
9. Suche nun im Disparitätenbild die vorhandenen Disparitäten. Verlängere diese Disparitäten bis zur nächsten Disparität. Mach dies einmal von links nach rechts und umgekehrt.
10. Wenn der Disparitätenunterschied zwischen den in Schritt 9 genannten Verlängerungen nicht zwischen 150 und 255 liegt, dann ist der Disparitätenwert aus dem Disparitätenbild zu nehmen, welches von links nach rechts verlängert wurde, ansonsten der andere.
11. Für jeden Bildpunkt muss nun die folgende Differenz erstellt werden: 255 – Disparitätenwert.
12. Nun kann das Bild ausgegeben werden.

Als Diagramm sieht dies folgendermaßen aus:

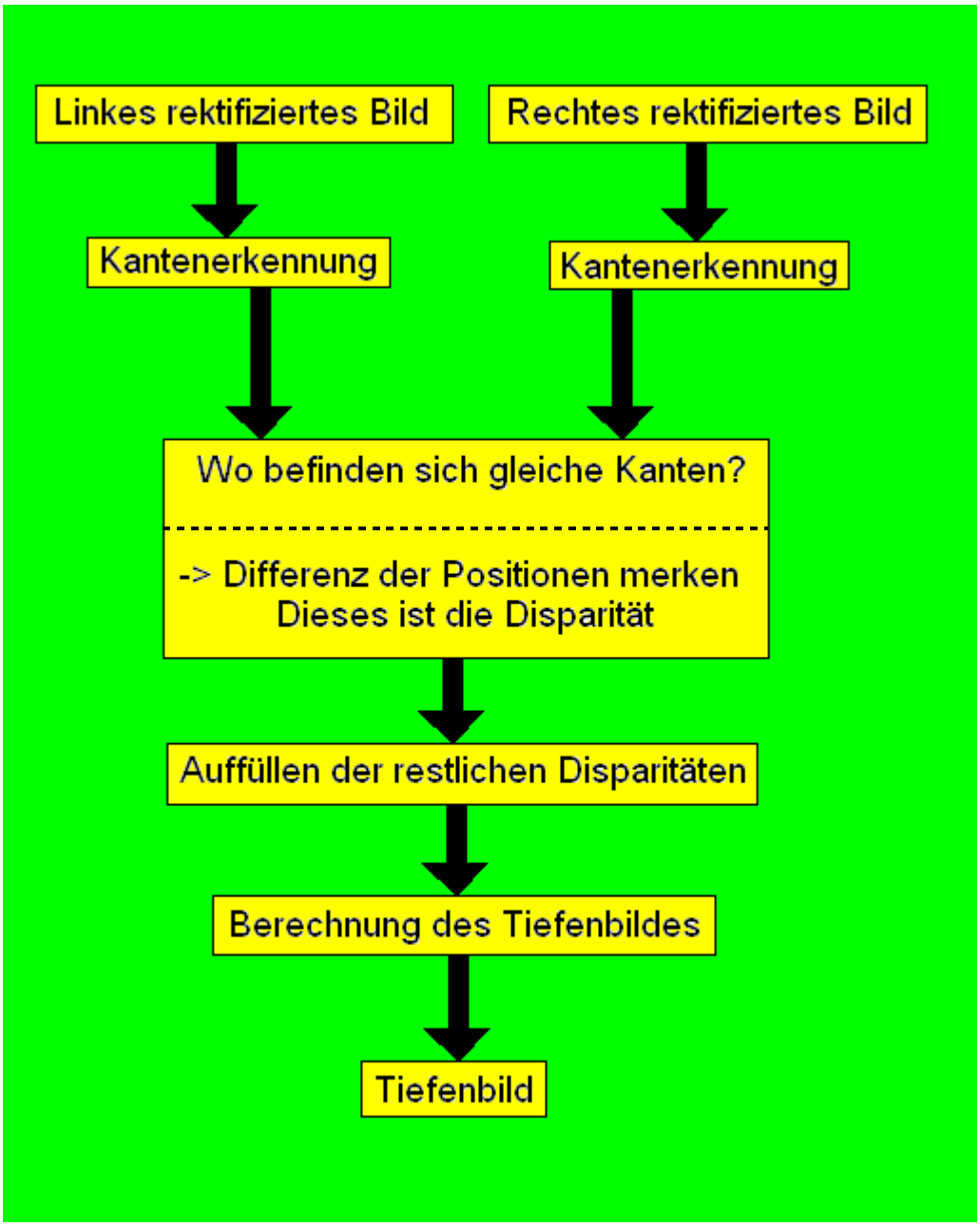


Abbildung 6.1 Tiefenbildalgorithmus Tobias Pfeiffer

6.2 Implementation des Algorithmus als Sourcecode

```
bool CStereoTiefenbilder2::Apply()
{
        source_left.castFrom(*m_chnsourceleft);         //Casten der Channel auf Images
        source_right.castFrom(*m_chnsourceright);

        int rows = m_chnsourceleft->rows(); //Speichern der Zeilen und Spaltenanzahl
        int columns = m_chnsourceleft->columns();
        destination.resize(rows,columns);  //Anpassen des zu verarbeitenden Bilder
        destination.empty();
        lti::image     verschiebungsarray, verschiebungsarray3;
        verschiebungsarray.resize(rows,columns);
        verschiebungsarray3.resize(rows,columns);

        for(int column=0; column<columns; column++)  //Nullifizieren
```

```
        for (int row=0; row<rows; row++){
                destination.at(row,column)=255;
                verschiebungsarray.at(row,column)=0;
                verschiebungsarray3.at(row,column)=0;
        }
// Kantendetektion
lti::channel8      chn8_left, chn8_right, ecken, chn8_left2, chn8_right2, chn8_left3, chn8_right3;
ecken.resize(rows,columns);

for (int row=0; row<rows; row++)  //Setzen des Kantenbildes auf 0
        for (int column=0; column<columns; column++)
                ecken.at(row,column)=0;

edgeDetector.apply(*m_chnsourceleft,chn8_left);
edgeDetector.apply(*m_chnsourceright,chn8_right);
edgeDetector2.apply(*m_chnsourceleft,chn8_left2);
edgeDetector2.apply(*m_chnsourceright,chn8_right2);
edgeDetector3.apply(*m_chnsourceleft,chn8_left3);
edgeDetector3.apply(*m_chnsourceright,chn8_right3);
// Verschiebungserrechnung für Ecken
for (int row=0; row<rows; row++)
        for (int column=0; column<columns; column++) {
                for (int verschiebung=0; verschiebung+column<columns; verschiebung++) {
                        if (((chn8_left.at(row,column+verschiebung)!=0) &&(chn8_right.at(row,column)!=0))||
                        ((chn8_left2.at(row,column+verschiebung)!=0) && (chn8_right2.at(row,column)!=0))||
                        ((chn8_left3.at(row,column+verschiebung)!=0) && (chn8_right3.at(row,column)!=0))) {
                                verschiebungsarray.at(row,column)=verschiebung;
                                verschiebungsarray3.at(row,column)=verschiebung;
                                ecken.at(row,column)=255;
                                break;
                        }
                        else {
                                ecken.at(row,column)=0;
                                verschiebungsarray.at(row,column)=0;
                                verschiebungsarray3.at(row,column)=0;
                        }
                }
        }

int delta = verschiebungsarray.at(0,0).getValue();   //Auffüllen von links nach rechts
for (int row=0; row<rows; row++)
        for (int column=0; column<columns; column++) {
                if (ecken.at(row,column)!=0) {
                        delta = verschiebungsarray.at(row,column).getValue();
                }
                else {
                        verschiebungsarray.at(row,column)=delta;
                }
        }

delta = verschiebungsarray3.at(rows-1,columns-1).getValue();  //Auffüllen von rechts nach links
for (int row = rows-1; row>0; row--)
        for (int column = columns-1; column>0; column--) {
                if (ecken.at(row,column)!=0) {
                        delta = verschiebungsarray3.at(row,column).getValue();
                }
                else {
                        verschiebungsarray3.at(row,column)=delta;
                }
        }

for (int row=0; row<rows; row++)   //Ausgleich der fehlinterpretierten Kanten
        for (int column=0; column<columns; column++) {
                int differenz = verschiebungsarray.at(row,column).getValue()-
                                verschiebungsarray3.at(row,column).getValue();
                if (differenz<=0) differenz*=(-1);
                if (150<=differenz<=255) verschiebungsarray.at(row,column) =
```

```
                                        verschiebungsarray3.at(row,column).getValue();
                }

        for (int row=0; row<rows; row++)   //Errechnen des Tiefenbildes
                for (int column=0; column<columns; column++) {
                        int disparitaet;
                        if ((verschiebungsarray.at(row,column).getValue()!=0)) {
                                disparitaet = BASELINE * BRENNWEITE /
                                                verschiebungsarray.at(row,column).getValue();
                                destination.at(row,column)= 255-disparitaet;
                        }
                }

        m_chndstTiefe.castFrom(destination);

        return true;
}
```

6.3 Beispielbild

Abbildung 6.2 Ausgangsbild Flur HAW [pfeiffer]

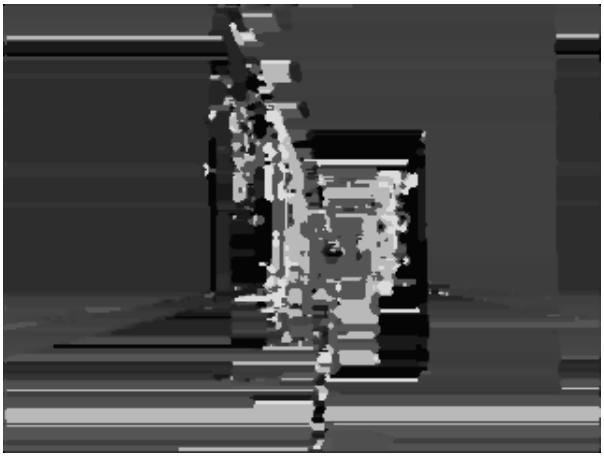


Abbildung 6.3 Tiefenbild Flur HAW [pfeiffer]

6.4 Versuchsaufbau

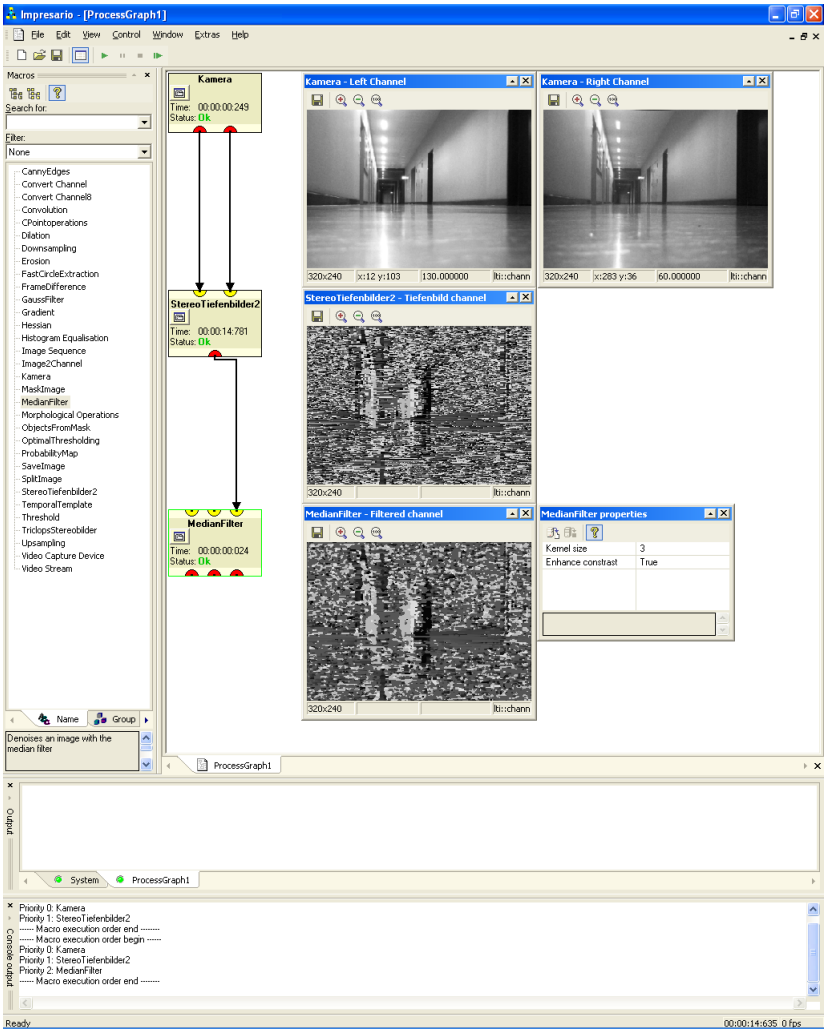


Abbildung 6.4 Versuchsaufbau Eigener Algorithmus [pfeiffer]

6.5 Interpretation der Ergebnisse

Der von mir entwickelte Algorithmus sollte in der Theorie gut laufen. Leider bestehen zwischen Theorie und Praxis doch häufig sehr große Unterschiede.

Die durch diesen Algorithmus erzeugten Bilder weisen im Prinzip zwar Ansätze für die Erstellung von Tiefenbildern auf, jedoch sind in den durch den Algorithmus erzeugten Bildern noch sehr viele Fehler und nicht behandelte Ausreißer vorhanden. Auch wenn das in Abbildung 6.3 [pfeiffer] gezeigte Bild einen Tiefeneffekt hat, stören die Fehler in den äußeren Bereichen des Bildes.

Bei folgendem theoretischen Bild versagt dieser Algorithmus vollkommen.

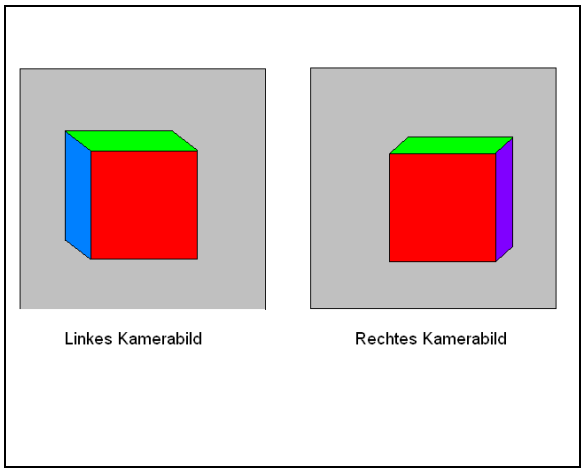


Abbildung 6.5 Theoretischer Würfel [pfeiffer]

In Abbildung 6.5 [pfeiffer] ist ein theoretischer Würfel zu sehen, wie er auch von einer Stereokamera aufgenommen werden könnte. Zwischen dem linken und dem rechten Kamerabild bestehen perspektivische Unterschiede. Im linken Bild zieht sich die Tiefe des Würfels nach links hinten. Im rechten Bild zieht sich die Tiefe des Würfels nach rechts hinten. Dadurch, dass der von mir entwickelte Algorithmus nur die Kanten sucht und anhand der gefundenen Kanten, welche der Algorithmus von links nach rechts vergleicht, würde der Algorithmus die linke Kante der blauen Fläche im linken Bild mit der linken Kante der roten Fläche im rechten Bild gleich setzen. Dieses ist jedoch vollkommen falsch.

Zudem werden perspektivische Fläche, wie die blaue beziehungsweise violette Fläche des theoretischen Würfels durch meinen Algorithmus als eine einheitliche Entfernung dargestellt. Es erfolgt also keine Anpassung der Entfernung je weiter eine homogene Fläche in den Hintergrund gerät.

Zusammenfassend gesehen ist der von mir entwickelte Algorithmus nicht in der Lage korrekte Tiefenbilder zu erzeugen. Es sind zwar Ansätze vorhanden, Tiefenbilder zu erzeugen, jedoch sind diese nicht tief greifend genug um korrekte Tiefenbilder zu erstellen. Der von mir entwickelte Algorithmus hat auch Schwierigkeiten mit unter- und überbelichteten Bildern. Um einigermaßen brauchbare Bilder zu erhalten, muss der aufgenommene Bildbereich, welcher in ein Tiefenbild umgewandelt werden soll, optimal Beleuchtet sein.

Aus diesem Grunde habe ich mich entschlossen, den Algorithmus von Point Grey Research in ein Impresario Modul umzusetzen und damit die Lösung von Point Grey für Impresario zur Verfügung zu stellen. Mehr dazu im nächsten Kapitel.

Kapitel 7

Algorithmus nach Point Grey Research

7.1 Beschreibung des Algorithmus

Die Umsetzung dieses Moduls für Impresario ist eine Kombination des Grabbens der Bilder und der eigentlichen Tiefenbildberechnung. Es existiert also nur ein einziges Modul für Impresario.

Die Initialisierung der Kamera und das Beenden der Point Grey Devices erfolgt wie in Kapitel 5 beschrieben. Aus diesem Grunde werde ich in diesem Kapitel nicht noch einmal auf diese Dinge eingehen. Interessierte Leser können sich in Kapitel 5 über diese Vorgänge informieren.

Nun aber zum eigentlichen Algorithmus. Zunächst werden die Bilder, wie in Kapitel 5 gegrabbt. Über ein Drop-Down Menü kann der Benutzer nun unter verschiedenen Tiefenbildoptimierungen auswählen. Die Unterschiede zwischen diesen Optimierungen sind unterschiedliche Einstellungen der Suchmaske für den „Sum of Absolute Difference“-Algorithmus sowie verschiedene Arten zum Validieren der Bilder. Die einzelnen Optionen zum Optimieren der Bilder sind wie folgt:

1. Default: Dies ist die Grundeinstellung für Tiefenbilder. Es werden folgende Einstellungen vorgenommen:
   Suchmaskengröße: 0
   Surface Validation: Ausgeschaltet
   Uniqueness Validation: Ausgeschaltet
   Texture Validation: Ausgeschaltet
   Back Forth Validation: Ausgeschaltet
2. Big Mask: Hier wird eine große Suchmaskengröße als Grundlage für die Tiefenbilder genommen. Die Einstellungen sind:
   Suchmaskengröße: 21
   Surface Validation: Ausgeschaltet
   Uniqueness Validation: Ausgeschaltet
   Texture Validation: Ausgeschaltet
   Back Forth Validation: Ausgeschaltet
3. Small Mask: Im Gegensatz zu Big Mask wird hier eine kleine Suchmaskengröße als Grundlage für die Tiefenbilder genommen. Die Einstellungen sind:
   Suchmaskengröße: 5
   Surface Validation: Ausgeschaltet

Uniqueness Validation: Ausgeschaltet
Texture Validation: Ausgeschaltet
Back Forth Validation: Ausgeschaltet

4. No Validation: Im Gegensatz zu Big Mask und Small Mask wird hier eine mittlere Suchmaskengröße als Grundlage für die Tiefenbilder genommen. Die Einstellungen sind:
   Suchmaskengröße: 11
   Surface Validation: Ausgeschaltet
   Uniqueness Validation: Ausgeschaltet
   Texture Validation: Ausgeschaltet
   Back Forth Validation: Ausgeschaltet
5. Surf-Tex-Val: Hier ist ebenfalls eine mittlere Suchmaskengröße eingestellt. Die restlichen Einstellungen lauten wie folgt:
   Suchmaskengröße: 11
   Surface Validation: Eingeschaltet
   Uniqueness Validation: Ausgeschaltet
   Texture Validation: Eingeschaltet
   Back Forth Validation: Ausgeschaltet
6. Back Forth: Die Einstellungen sind wie bei Surf-Tex-Val, jedoch ist das Validieren anders eingestellt. Die Einstellungen sind:
   Suchmaskengröße: 11
   Surface Validation: Eingeschaltet
   Uniqueness Validation: Ausgeschaltet
   Texture Validation: Eingeschaltet
   Back Forth Validation: Eingeschaltet

Im nächsten Schritt werden die Bilder nach dem Point Grey eigenen Verfahren rektifiziert. Nach diesem Schritt bearbeitet der Point Grey eigene Stereoalgorithmus die Bilder. Bei diesem Algorithmus werden die Kanten in den Bildern erkannt und aus Ihnen die Disparitäten für das gesamte Bild berechnet. Im nächsten Schritt folgt die Vereinheitlichung der Disparitäten. Disparitäten, die nicht korrekt erscheinen werden hier auf Schwarz gesetzt. Als letzter Schritt werden nun noch das Referenzbild und das Tiefenbild auf die Ausgänge des Moduls gelegt.

7.2 Implementation des Algorithmus als Sourcecode

Wie bereits in Kapitel 7.1 erwähnt erfolgt die Initialisierung der Kamera und das Beenden der Point Grey Devices auf die Gleiche Art und Weise wie sie in Kapitel 5 beschrieben worden ist. Aus diesem Grunde wird hier nur der Hauptteil des Moduls als Sourcecode ohne Fehlerbehandlung aufgelistet.

```
bool CTriclopsStereobilder::Apply()
{
        printf( "Grabbing an image...\n" );   //Grabben der Bilder von der Kamera
        e = digiclopsGrabImage( digiclops );

        int r,c;  //Erstellen von Integervariablen für Reihen- und Spaltenzähler
        e = digiclopsExtractTriclopsInput( digiclops, STEREO_IMAGE, &triclopsInput );//Extrahieren TriclopsInput
        digiclopsGetTriclopsContextFromCamera(digiclops,&triclops); //Erstellen des Triclops-Containers
```

```
// Hier nun die Optionauswahl:
// 0 = Default, 1 = Big Mask, 2 = Small Mask, 3 = No Validation, 4 = Surf-Tex-Val, 5 = Back Forth

switch (option)    {
                   case 0:   {          //Default
                             triclopsSetStereoMask(triclops,0);
                             triclopsSetSurfaceValidation(triclops,0);
                             triclopsSetUniquenessValidation(triclops,0);
                             triclopsSetTextureValidation(triclops,0);
                             triclopsSetBackForthValidation(triclops,0);
                             break;
                             }
                   case 1:   {          //Big Mask
                             triclopsSetStereoMask(triclops,21);
                             triclopsSetSurfaceValidation(triclops,0);
                             triclopsSetUniquenessValidation(triclops,0);
                             triclopsSetTextureValidation(triclops,0);
                             triclopsSetBackForthValidation(triclops,0);
                             break;
                             }
                   case 2:   {          //Small Mask
                             triclopsSetStereoMask(triclops,5);
                             triclopsSetSurfaceValidation(triclops,0);
                             triclopsSetUniquenessValidation(triclops,0);
                             triclopsSetTextureValidation(triclops,0);
                             triclopsSetBackForthValidation(triclops,0);
                             break;
                             }
                   case 3:   {          //No Validation
                             triclopsSetStereoMask(triclops,11);
                             triclopsSetSurfaceValidation(triclops,0);
                             triclopsSetUniquenessValidation(triclops,0);
                             triclopsSetTextureValidation(triclops,0);
                             triclopsSetBackForthValidation(triclops,0);
                             break;
                             }
                   case 4:   {          //Surf-Tex-Val
                             triclopsSetStereoMask(triclops,11);
                             triclopsSetSurfaceValidation(triclops,1);
                             triclopsSetUniquenessValidation(triclops,0);
                             triclopsSetTextureValidation(triclops,1);
                             triclopsSetBackForthValidation(triclops,0);
                             break;
                             }
                   case 5:   {          //Back Forth
                             triclopsSetStereoMask(triclops,11);
                             triclopsSetSurfaceValidation(triclops,1);
                             triclopsSetUniquenessValidation(triclops,0);
                             triclopsSetTextureValidation(triclops,1);
                             triclopsSetBackForthValidation(triclops,1);
                             break;
                             }
                   }

// Rektifizieren der Bilder
triclopsRectify(triclops,&triclopsInput);
// Anwendung Stereoalgorithmus
triclopsStereo(triclops);
// Holen des Bildes
triclopsGetImage(triclops,TriImg_DISPARITY,TriCam_REFERENCE,&triImage);

//scaleImage(&triImage,90,255);
double dMinOut = (double) 90;
double dMaxOut = (double) 255;
double dMaxDisp = 0;
double dMinDisp = 255;
```

```
		for (r=0; r< triImage.nrows; r++) {  //Finden der minimalen und maximalen Disparität
			unsigned char*	pucSrc = triImage.data + r * triImage.rowinc;
			for (c=0; c<triImage.ncols; c++) {
				if (pucSrc[c]<240) {
					double dDisp = (double) pucSrc[c];
					if (dMaxDisp<dDisp) dMaxDisp = dDisp;
					if (dMinDisp>dDisp) dMinDisp = dDisp;
				}
			}

		}

		for (r=0; r<triImage.nrows; r++) {  //Ausgleich der Disparität
			unsigned char* pucSrc = triImage.data + r * triImage.rowinc;
			for (c=0; c<triImage.ncols; c++) {
				if (pucSrc[c]<240) {
					double dDisp = (double) pucSrc[c];
					double dOut = (dDisp-dMinDisp) * (dMaxOut-dMinOut) / (dMaxDisp-dMinDisp);
					dOut += dMinOut;
					pucSrc[c] = (unsigned char) dOut;
				}
				else {
					pucSrc[c] = 0;
				}
			}
		}

		// Von einem Bild ins andere kopieren
		m_depthImage.resize(triImage.nrows,triImage.ncols);
		for (r=0; r<triImage.nrows; r++) {
			unsigned char* pucSrc = triImage.data + r * triImage.rowinc;
			for (c=0; c<triImage.ncols; c++) {
				m_depthImage.at(r,c) = pucSrc[c];
			}
		}

		//Erstellen von Referenzbild
		triclopsGetImage(triclops,TriImg_RECTIFIED,TriCam_REFERENCE,&digiImage);
		m_depthImage_left.resize(digiImage.nrows,digiImage.ncols);
		for (r=0; r<digiImage.nrows; r++) {
			unsigned char* pucSrc = digiImage.data + r * digiImage.rowinc;
			for (c=0; c<digiImage.ncols; c++) {
				m_depthImage_left.at(r,c) = pucSrc[c];
			}
		}
		return true;
}
bool CTriclopsStereobilder::ExitMacro() { //Siehe Kapitel 5
}
void CTriclopsStereobilder::ParameterChanged(int nParameter)
{
		switch(nParameter)		{
						case 0:	{
							std::string	strOperationType = GetParameterValue<std::string>(0);
							if (strOperationType == "Default")			option=0;
							else if (strOperationType == "Big Mask")		option=1;
							else if (strOperationType == "Small Mask")		option=2;
							else if (strOperationType == "No Validation")		option=3;
							else if (strOperationType == "Surf-Tex-Val")		option=4;
							else if (strOperationType == "Back Forth")		option=5;
							else 						option=0;
							break;
							}
						}
}
```

7.3 Beispielbilder

Abbildung 7.1 Referenzbild [pfeiffer]

Abbildung 7.2 Default [pfeiffer]

Abbildung 7.3 Big Mask [pfeiffer]

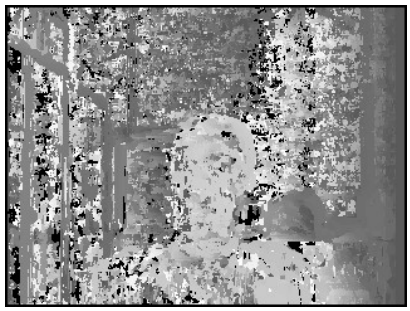


Abbildung 7.4 Small Mask [pfeiffer]

Abbildung 7.5 No Validation [pfeiffer]

Abbildung 7.6 Surf-Tex-Val [pfeiffer]

Abbildung 7.7 Back Forth [pfeiffer]

7.4 Versuchsaufbau

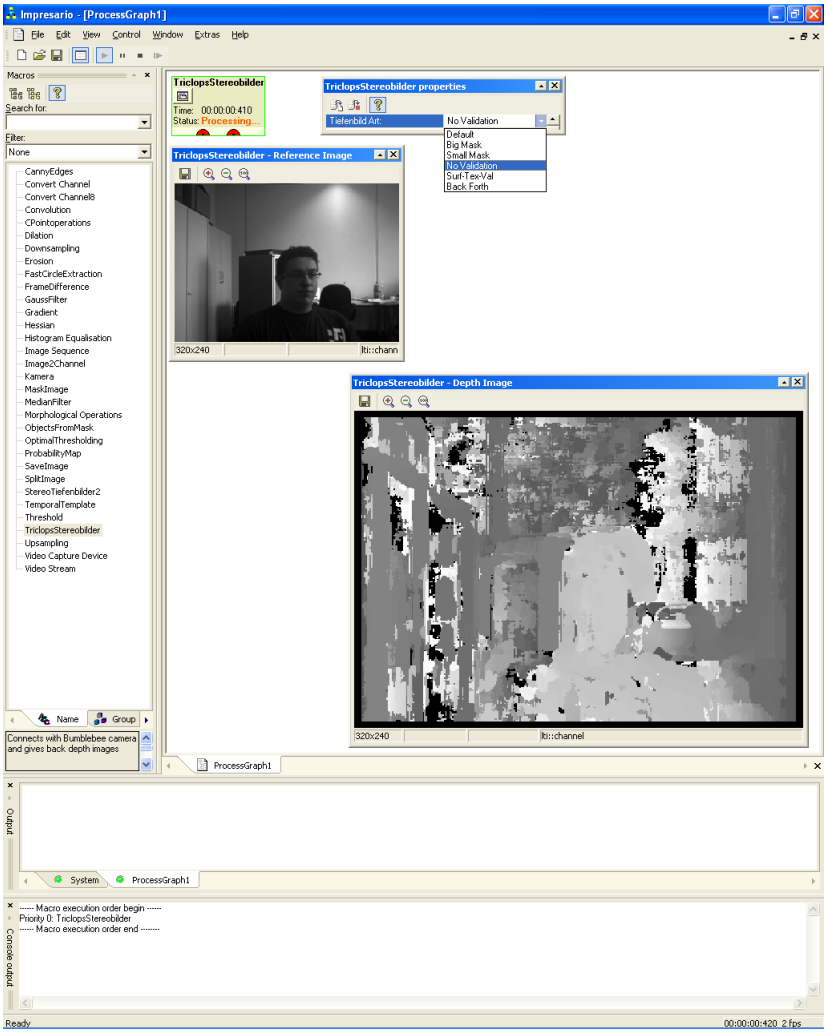


Abbildung 7.8 Versuchsaufbau Point Grey [pfeiffer]

7.5 Interpretation der Ergebnisse

An den in Kapitel 7.3 gezeigten Bildern ist die Qualität der Point Grey eigenen Algorithmen erkennbar. Hier treten die Qualitätsunterschiede zu dem von mir entwickelten Algorithmus klar hervor. Es wird - allein nur durch Betrachten der Bilder – klar, welche Auswirkungen die unterschiedlichen Einstellungen auf die resultierenden Tiefenbilder haben. Je nachdem, welche Einstellung gewählt wird, umso besser oder schlechter fallen die Ergebnisse des resultierenden Tiefenbildes aus.

Bei der Default - Einstellung wird das Tiefenbild so interpretiert, wie es die Stereo-Funktion von Point Grey erhält. Es ist einfach gehalten ohne Korrekturen an Disparitäten. Bei der Big Mask – Einstellung wird dem Sum of Absolute Difference – Algorithmus eine große Suchmaske vorgegeben. Durch diese große Maske wirkt das Bild sehr rund und ausgeglichen. Bei der Small Mask – Einstellung wird eine entsprechend kleinere Suchmaske dem Sum of Absolute Difference – Algorithmus übergeben. Das resultierende Bild beinhaltet mehr Details, lässt aber auch ein Rauschen, welches vielen Tiefenbildalgorithmen gemein ist, durchdringen. Die No Validation – Einstellung liefert in der Regel die besten Ergebnisse von den verfügbaren Optionen. Bei dieser Einstellung wird, wie auch bei den folgenden Einstel-

lungen, dem Sum of Absolute Difference – Algorithmus eine mittelgroße Suchmaske übergeben. Die Einstellung filtert das Rauschen ein wenig heraus. Auch werden kleinere Fehler geglättet. Die Surf-Tex-Val – und die Back Forth – Einstellung haben die Gemeinsamkeit, dass beide nur mit den erkannten Kanten arbeiten und nicht versuchen den Rest des Bildes zu interpretieren. Dieses hat den Vorteil, dass somit nur die wirklichen Disparitäten zum Vorschein kommen. Bei der Surf-Tex-Val – Einstellung werden die Disparitäten von sämtlichen gefundenen Kanten für die Tiefenberechnung interpretiert. Bei der Back Forth – Einstellung hingegen werden Kanten ausgemustert, die nach dem Algorithmus von Point Grey nicht in die Disparitätenkarte passen. In beiden Einstellungen wird versucht, das lästige Rauschen der übrigen Einstellmöglichkeiten einzudämmen. Ansonsten sind beide Einstellungen sehr ähnlich.

Leider hat dieser Algorithmus auch seine eigenen Darstellungsfehler. Das Rauschen, welches viele Tiefenalgorithmen gemein haben, wird hier auch nicht richtig beseitigt. Bei schlechter Beleuchtung der Bildmotive tritt somit auch ein Rauschen auf, welches die Tiefenbildberechnung stark beeinträchtigt. In solchen Fällen sind die Surf-Tex-Val - und die Back Forth – Einstellung die besten Optionen zur Erstellung von Tiefenbildern. Bei optimaler Beleuchtung sollte dieses Rauschen weniger werden. Dieses konnte leider nicht geprüft werden, weil die entsprechenden Beleuchtungsmittel nicht zur Verfügung stehen um eine optimale Beleuchtung der Bildmotive zu gewährleisten. Zudem können bei diesem hier vorliegenden Modul keine eigenen oder fremde Stereobilder dem Algorithmus zugeführt werden.

Das größte Problem ist allerdings der große Bedarf an Arbeitspeicher. Obwohl der Arbeitsrechner mit 1 GB Speicher bestückt ist, ist dieser Speicher sehr schnell ausgeschöpft. Dadurch, dass der Speicher schnell zu 100 % erschöpft ist und das Programm versucht auf noch mehr Speicher zuzugreifen, gibt es regelmäßig Abstürze wegen illegalen Speicherzugriffs.

Kapitel 8

Fazit und Blick in die Zukunft

8.1 Fazit

Die Beschäftigung mit dem Thema der Tiefenbildverarbeitung hat mir viele neue Erkenntnisse in diesem Bereich gebracht. Auch wenn mein Algorithmus nur ansatzweise Tiefenbilder erzeugen kann, gibt es dennoch die Möglichkeiten Tiefenbilder zu erzeugen. Erst bei der Umsetzung des Algorithmus von Point Grey Research wurden mir die Möglichkeiten klar, wie die Point Grey eigenen Bildformate auf die Impresario eigenen Bildformate übertragen werden können. Leider klappt dies bislang nur in eine Richtung, nämlich von Point Grey nach Impresario. Ich konnte leider keinen umgekehrten Weg finden, da Point Grey in seinen Bildern ganz anders vorgeht als Impresario. In Impresario sind die einzelnen Pixel durch Angabe des Reihen- beziehungsweise Spaltenindex direkt ansprechbar. Dieses einfache Prinzip verwendet Point Grey leider nicht. Point Grey speichert seine Bilder reihenweise als 'unsigned char' – Pointer ab. Um an die einzelnen Reihen zu gelangen, wird bei Point Grey eine Variable namens 'rowinc' gesetzt. 'rowinc' bezeichnet wie viele Bytes in einer Reihe gespeichert werden. Nur über dieses 'rowinc' kann dann auf die einzelnen Bildpunkte zugegriffen werden. Dadurch, dass ich nicht heraus finden konnte wie diese Variable gesetzt werden muss, konnte ich den Weg vom Impresario – Bild in ein Point Grey – Bild nicht realisieren. Durch diesen Umstand war es leider nicht möglich den Algorithmus von Point Grey in ein externes Modul zu verlagern. Vielleicht wird dieser Weg ja einmal in einer späteren Arbeit erforscht, da die restliche verbleibende Zeit - um diese Arbeit zu vollenden - nicht ausreicht, dieses Thema intensiver zu betrachten.

Der Algorithmus von Point Grey liefert gute Ergebnisse, auch wenn bei vielen Bildern ein Grundrauschen vorhanden ist, welches allerdings die meisten Tiefenalgorithmen auch haben. In den Bildern von Point Grey sind Objekte, welche sich nahe an der Kamera befinden, heller dargestellt als Objekte, welche weiter entfernt sind. Hierdurch wird der Tiefenbildcharakter vermittelt. Auch durch die sehr kurze Bearbeitungszeit ist der Point Grey – Algorithmus sehr effizient. Die Bearbeitungszeit liegt bei Point Grey im Bereich von Sekunden. Dieses ist ein sehr gutes Ergebnis. Mein eigener Algorithmus benötigt bereits mehr als 20 Sekunden pro Bild bei einem schlechteren Ergebnis.

Leider haben Bildbearbeitungen dieser Art einen sehr großen Bedarf an Arbeitsspeicher. Der Speicher von 1 GB in dem Arbeitsrechner, auf dem die Entwicklung geschehen ist, ist sehr schnell erschöpft und illegale Speicherzugriffe führen zum

Programmabsturz.

Wie auch schon in den vorherigen Kapiteln erwähnt, führt eine schlechte Beleuchtung der Bildmotive zu einem starken Rauschen in den Tiefenbildern. Auch wenn die meisten Tiefenalgorithmen dieses Rauschen auch haben, ist es doch sehr störend und kann zu Fehlinterpretationen der Bilder führen. Mit zunehmend optimaler Beleuchtung sollte dieses Rauschen weniger werden und bessere Tiefenbilder hervor bringen.

8.2 Blick in die Zukunft

Die hier umgesetzten Algorithmen zeigen deutlich, dass die Forschung in Sachen Tiefenbilder noch in den Kinderschuhen steckt. Es werden aber zu diesem Zeitpunkt schon sehr gute Ergebnisse erzielt, welche sich mit der Verbesserung von Hard- und Software sicherlich in Zukunft noch steigern lassen.

Zu dem Zeitpunkt der Entstehung dieser Arbeit war die LTI-Lib 1.9 die aktuelle Bibliothek für Impresario. Die LTI-Lib 2 steht zwar schon in den Startlöchern, wurde aber noch nicht für den öffentlichen Gebrauch freigegeben. Laut einiger Kommentare in Foren soll der Zugriff auf Firewire - Kameras in der Version 2 hinzu kommen. Vielleicht gelingt es dann den Point Grey Algorithmus anzuwenden.

Mit der Verbesserung der Kameras ist auch zu rechnen. Gerade ein besserer Helligkeitsausgleich der Kameras wäre sehr wünschenswert, um das störende Rauschen bei den mit aktuellen Kameras erstellten Bildern zu reduzieren. Point Grey hat mittlerweile eine verbesserte Version der 2 – Linsen – Kamera Bumblebee auf den Markt gebracht. Leider kann ich hier nichts über die Helligkeitssteuerung dieser verbesserten Version sagen.

Der Bereich der Tiefenbilderkennung ist in Zukunft ein sehr gefragtes Thema. Dieses ist zum Beispiel bei der Automobilindustrie der Fall. Hier ist die Entwicklung von elektronischen Einparkhilfen, die auch theoretisch autonom ein Auto einparken könnten, bereits dabei Realität zu werden. Auch die Entwicklung von automatischen Systemen, welche einen Sicherheitsabstand zu voraus fahrenden Fahrzeugen einhalten, könnte in Zukunft von der Tiefenbilderkennung profitieren.

Wie gesagt steckt die Entwicklung von Tiefenbildalgorithmen noch in den Kinderschuhen. Hier wird es in Zukunft noch viel zu erforschen geben.

Anhang Tiefenbilder

Die in diesem Kapitel dargestellten Bilder wurden zum großen Teil mit Hilfe einer speziellen Lampe für Foto- beziehungsweise Filmaufnahmen erstellt. Bei der Lampe handelt es sich um die Nizolux 1000 G mit Gebläse von Braun. Dieses ist eine kompakte Film- und Foto-Sicherheitsleuchte mit 1000 Watt Halogenlampe.

Da das rechte und das linke Ursprungsbild sehr ähnlich sind, ist in diesem Kapitel nur immer eines der beiden Bilder abgebildet. Durch die schlechten Ergebnisse des von mir entwickelten Algorithmus werden hier nur die Tiefenbilder des Point Grey Algorithmus gezeigt.

Anhang Tiefenbilder: Arbeitsraum

Abbildung A.1 Arbeitsraum [pfeiffer]

Abbildung A.2 Tiefenbild Arbeitsraum

Im Tiefenbild ist deutlich zu erkennen, dass der Versuchsschrank durch seine hellere Färbung in den Vordergrund tritt. Auch der Schrank am linken Bildrand tritt deutlich durch seine dunklere Verfärbung, je tiefer er im Raum steht, hervor.

Anhang Tiefenbilder: Computer

In den nun folgenden Bilder ist eine Abbildung des Computers zu erkennen, an dem diese Arbeit entstanden ist.

Abbildung A.3 Computer [pfeiffer]

Abbildung A.4 Default

Abbildung A.5 Big Mask

Abbildung A.6 Small Mask

Abbildung A.7 No Validation

Abbildung A.8 Surf-Tex-Val

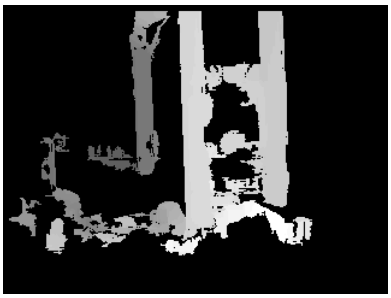


Abbildung A.9 Back Forth

Aus den Tiefenbildern des Computers geht hervor, dass die Einstellungen Surf-Tex-Val und Back Forth die besten Ergebnisse hinsichtlich des Rauschverhaltens in den Bildern bringen. Big Mask besticht hingegen durch seine groben aber doch runden Strukturen während Small Mask mehr Details zeigt, wie zum Beipiel die Strukturen des Computers.

Anhang Tiefenbilder: Mülleimer

Abbildung A.10 Mülleimer Links [pfeiffer]

Abbildung A.11 Mülleimer Rechts [pfeiffer]

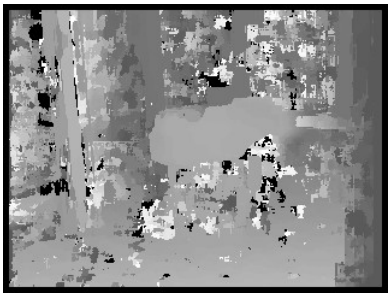


Abbildung A.12 Default

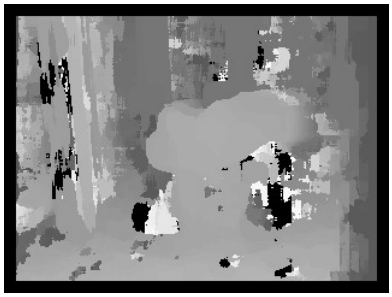


Abbildung A.13 Big Mask

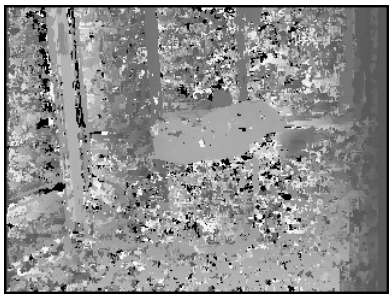


Abbildung A.14 Small Mask

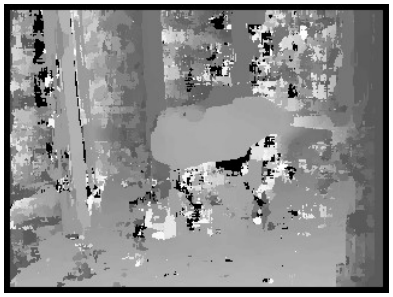


Abbildung A.15 No Validation

Abbildung A.16 Surf-Tex-Val

Abbildung A.17 Back Forth

Bei diesen Abbildungen ist auch das linke Kamerabild mit aufgeführt, um die Ähnlichkeiten der beiden Bilder zu belegen. Der Unterschied zwischen dem linken und dem rechten Kamerabild ist eine Verschiebung auf horizontaler Ebene.

Auch in diesen Bildern tritt wieder stark das Tiefenbildrauschen hervor.

Anhang Tiefenbilder: Eigenportrait

Abbildung A.18 Eigenportrait [pfeiffer]

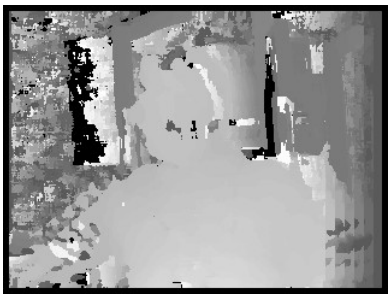


Abbildung A.19 Default

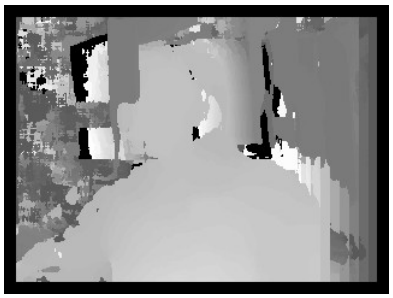


Abbildung A.20 Big Mask

Abbildung A.21 Small Mask

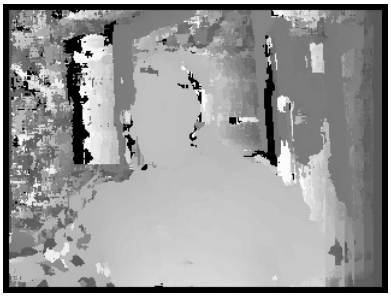


Abbildung A.22 No Validation

Abbildung A.23 Surf-Tex-Val

Abbildung A.24 Back Forth

In diesen Tiefenbildern ist erkennbar, dass Objekte, die sich nahe an den Objektiven der Kamera befinden besser erkannt werden als Objekte, die weiter entfernt sind. Bei diesen Bildern wird der Unterschied zwischen der Surf-Tex-Val und Back Forth deutlich. Während Surf-Tex-Val den Oberkörper der abgebildeten Person als eine einheitliche Fläche darstellt, wird bei Back Forth der Oberkörper genauer in der Tiefe dargestellt. No Validation liefert auch gute Ergebnisse. Bei dieser Aufnahme ist das Hintergrundrauschen deutlich reduziert. Small Mask zeigt gegenüber Big Mask mehr Details, dafür ist aber auch ein stärkeres Rauschen vorhanden.

Anhang Tiefenbilder: Flur in der HAW Hamburg

In den folgenden Bildern ist ein Auschnitt aus einem der Flure des Gebäudes, in dem diese Arbeit entstanden ist, zu sehen.

Abbildung A.25 Flur in der HAW Hamburg [pfeiffer]

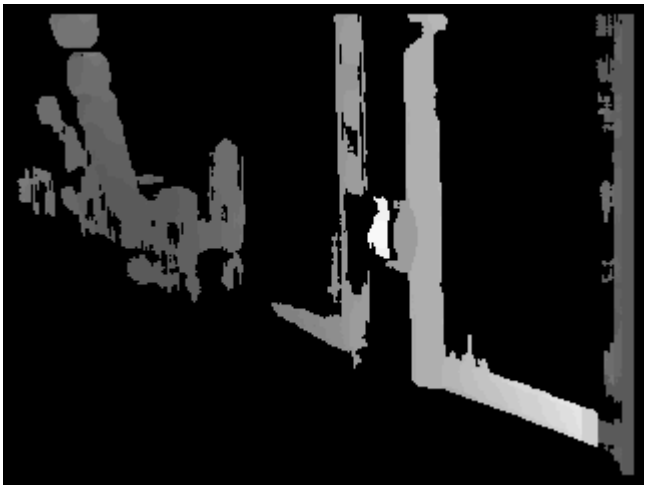


Abbildung A.26 Tiefenbild

Dieses Tiefenbild ist Resultat von Surf-Tex-Val. Durch die Unterbelichtung der Originalbilder ist in den anderen Tiefenbildern ein zu starkes Rauschen vorhanden, so dass dort kein brauchbares Tiefenbild entstanden ist. Aus dem hier abgebildeten Tiefenbild ist aber trotz dessen deutlich ein Tiefenverlauf in der Fußleiste zu erkennen.

Anhang Tiefenbilder: Eisteeflasche

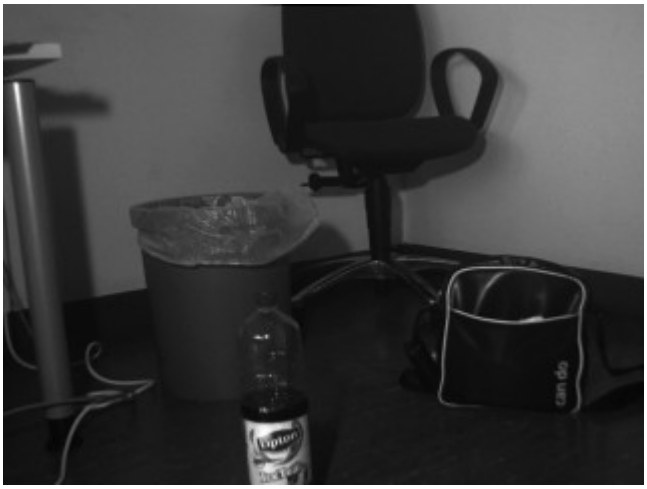


Abbildung A.27 Eisteeflasche [pfeiffer]

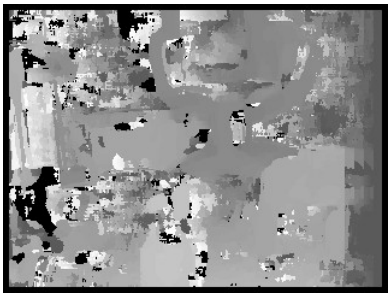


Abbildung A.28 Default

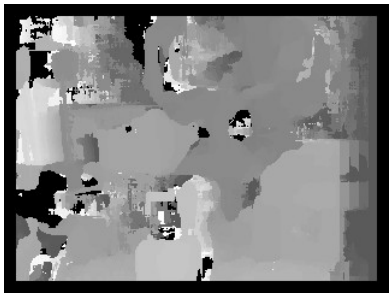


Abbildung A.29 Big Mask

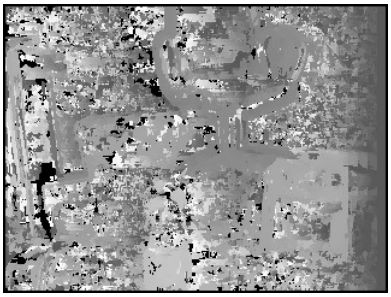


Abbildung A.30 Small Mask

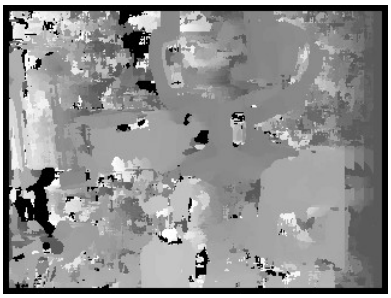


Abbildung A.31 No Validation

Abbildung A.32 Surf-Tex-Val

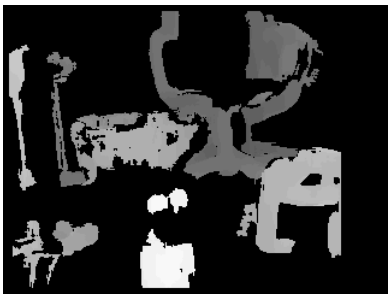


Abbildung A.33 Back Forth

In den hier gezeigten Tiefenbildern ist besonders das starke Rauschen bei Small Mask zu erkennen. Die restlichen Bilder sind jedoch gut als Tiefenbilder zu erkennen. No Validation hat, trotz des Rauschens, einen guten Tiefenbildcharakter. Trotz dessen sind bei Surf-Tex-Val und Back Forth immer noch die besten Ergebnisse vorzuweisen, da hier das Rauschen entfernt ist.

Anhang Tiefenbilder: Notausgang

In den nun folgenden Bildern wird eine Abbildung eines Notausganges in der HAW Hamburg dargestellt.

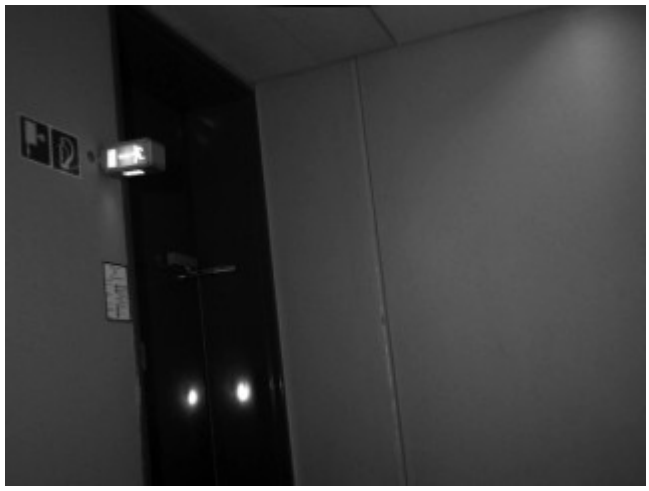


Abbildung A.34 Notausgang [pfeiffer]

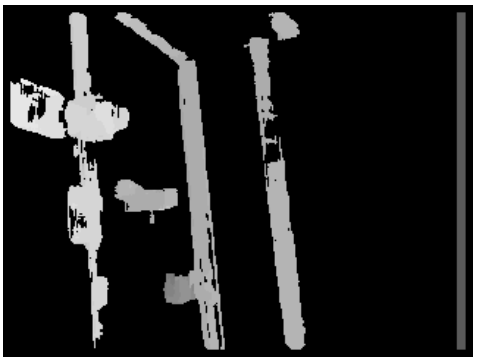


Abbildung A.35 Surf-Tex-Val

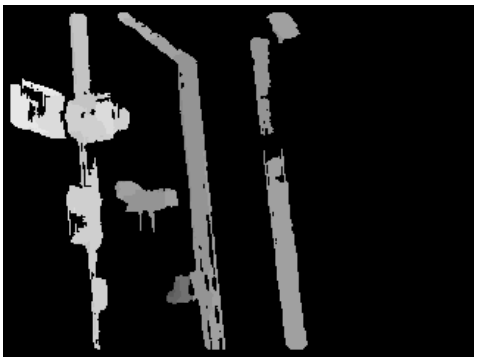


Abbildung A.36 Back Forth

Die hier gezeigten Tiefenbilder sollen die Ähnlichkeiten zwischen Surf-Tex-Val und Back Forth aufzeigen. Bis auf wenige Pixel sind beide Bilder gleich.

Anhang Tiefenbilder: Ausgang

Abbildung A.37 Ausgang [pfeiffer]

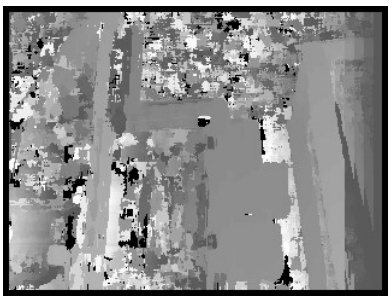


Abbildung A.38 Default

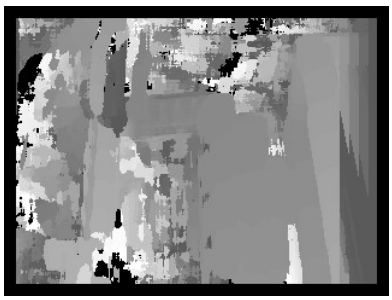


Abbildung A.39 Big Mask

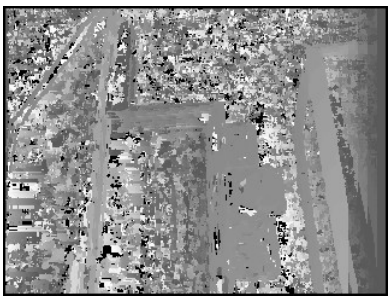


Abbildung A.40 Small Mask

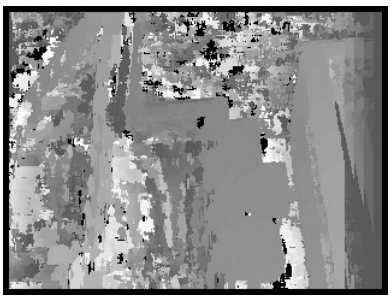


Abbildung A.41 No Validation

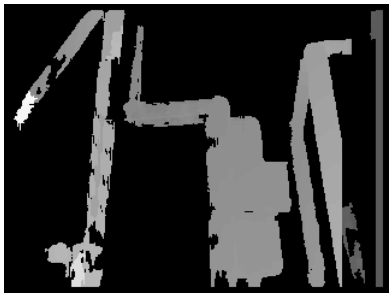


Abbildung A.42 Surf-Tex-Val

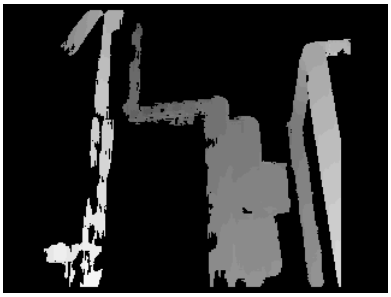


Abbildung A.43 Back Forth

Mit diesen Bildern der Tür des Raumes, in dem diese Arbeit entstanden ist, soll nun diese Arbeit beendet werden.

Inhalt der beiliegenden CD-ROM

Die beiliegende CD-ROM ist in mehrere Unterverzeichnisse gegliedert.

Verzeichnis	**Inhalt**
\arbeit\	Dieses Dokument als PDF
\source\	Verzeichnis der Quelldateien
\source\Kameratest\	Sourcecode Bildergrabber und Point Grey Lösung
\source\Tiefenbildalgorithmen\	Sourcecode Tiefenbildalgorithmus
\programme\	Einige der für diese Arbeit verwendeten Programme
\programme\perl\	Programmiersprache Perl
\programme\impressario\	Impresario
\programme\direct_x\	Direct X SDK

Abbildungsverzeichnis

Quellverzeichnis

[maya]	Computational Photography, Alexei Efros, CMU, Fall 2005, http://www.maya.com/local/limsc/courses/15463/Lectures/stereo.pdf
[wiki_stereo]	Wikipedia, Stereokamera, http://de.wikipedia.org/wiki/Stereokamera
[ptgrey]	Point Grey Research, http://www.ptgrey.com
[predrag]	Verdeckungsanalyse in Stereobildern anhand des Vergleiches eines kooperativen und eines bayes'schen Verfahrens, Predrag Steric, TU Wien, http://www.prip.tuwien.ac.at/Teaching/WS/ProSemSab/prosem03/files/predrag.mht
[pfeiffer]	Aufnahmen Tobias Pfeiffer mit Bumblebee Kamera, Hochschule für Angewandte Wissenschaften Hamburg
[impresario]	Impresario, RWTH Aachen, http://www.techinfo.rwth-aachen.de/Software/Impresario/
[ltilib]	LTI-Lib, RWTH Aachen, http://ltilib.sourceforge.net/doc/homepage/index.shtml
[irfan]	Irfan View, kostenloses Bildbetrachtungsprogramm, http://www.irfanview.com/
[meerbergen]	G. Van Meerbergen, M. Vergauven, M. Pollefeys, L. Van Gool; A hierarchical stereo algorithm using dynamic programming
[kux]	Michael Kux, Technische Universität Wien; Vergleich von Verdeckungsanalysen in der Stereo Vision
[triclops]	Point Grey Research, Triclops Software Developement Kit Manual Version 3.1
[bernardino]	Alexandre Bernardino und José Santos-Victor, Instituto Superior Técnico, Lissabon, Portugal; A Binocular Stereo Algorithm for Log-polar Foveated Systems
[thiel]	Christian Thiel, Universität Mannheim; Verdeckungsanalyse in Stereobildern unter Verwendung von Disparity-Space-Images und Dynamischer Programmierung, Seite 4
[bleyer]	Michael Bleyer und Margrit Gelautz, Technische Universität Wien; A LAYERED STEREO ALGORITHM USING IMAGE SEGMENTATION AND GLOBAL VISIBILITY CONSTRAINTS

Beiliegende CD-ROM

Versicherung über die Selbständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) PO2001 TI ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 30. August 2007

_____

Tobias Pfeiffer