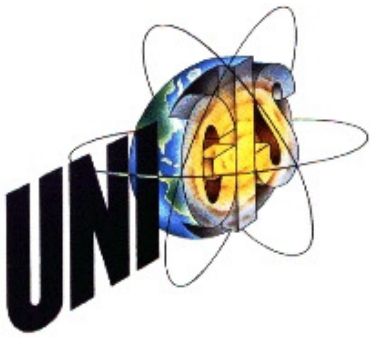


Master Thesis

im Rahmen des
Universitätslehrganges „Geographical Information Science & Systems“
(UNIGIS MSc) am Zentrum für GeoInformatik (Z_GIS)
der Paris Lodron-Universität Salzburg

zum Thema

„Problematik der Korrespondenz beim Matching von Drohnenluftbildern“

OpenCV-Lösungsansätze in der Praxis

vorgelegt von

B.Eng. Tino Winkelbauer
U1475, UNIGIS MSc Jahrgang 2011

Zur Erlangung des Grades
„Master of Science (Geographical Information Science & Systems) – MSc(GIS)”

Gutachter:
Ao. Univ. Prof. Dr. Josef Strobl

Stendal, 30.12.2013

Eidesstattliche Erklärung

Ich versichere, dass ich diese Arbeit selbständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und alle wörtlich und sinngemäß übernommenen Textstellen als solche kenntlich gemacht habe.

Ort, Datum

Unterschrift

Danksagung

Ich bedanke mich bei allen, die mich während meiner Zeit des Studiums in Salzburg und während der Zeit der Master Thesis unterstützt haben.

Insbesondere halfen mir dabei meine Arbeitskollegen der GEO-METRIK Ingenieurgesellschaft mbH Stendal, Karl Atzmanstorfer als Jahrgangsbetreuer des UNIGIS Salzburg und Harald Herda als wissenschaftlicher Mitarbeiter der Beuth Hochschule Berlin. Die Unterstützung durch Prof. Michael Breuer als Betreuer dieses Themas von der Beuth Hochschule erleichterte mir die Einarbeitung in das komplexe Thema. Die Treffen mit ihm halfen mir bei der Planung und Umsetzung der Master Thesis.

Besonderer Dank gilt an dieser Stelle meiner Frau, die mich in dieser Zeit mit Kraft, Gelassenheit und Ausdauer gestützt und gestärkt hat.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ANN	Approximate Nearest Neighbors
ASCII	American Standard Code for Information Interchange
BA	Bündelausgleichung
BRIEF	Binary Robust Independent Elementary Features
FAST	Features from Accelerated Segment Test
FREAK	Fast Retina Keypoint
DLL	Dynamic Link Library
DOM	digitales Oberflächenmodell
GCP	Ground Control Point
GFTT	Good Features to Track
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
IMU	Inertial Measurement Unit
INS	Inertial Navigation System
LPS	Leica Photogrammetry Suite
LWL	Lichtwellenleiter
MSER	Maximally Stable Extremal Regions
NIR	Nahes Infrarot
OpenCV	Open Source Computer Vision Library
ORB	Oriented FAST and Rotated BRIEF
OS	Operating System
RGB	Rot Grün Blau
SfM	Structure from Motion
SIFT	Scale Invariant Feature Transform

SURF	Speeded Up Robust Features
UAS	Unmanned Aircraft System
UAV	Unmanned Aerial Vehicle

1. Einleitung

„Die Hauptaufgabe der Luftbildvermessung ist eine dreidimensionale Erfassung der natürlichen und künstlichen Landschaft.“ [KRA04, S.144]

Diese kann sie vor allem durch die stetige Weiterentwicklung der Mikrotechnologie leisten, die in immer mehr Bereichen der Wissenschaft und Forschung Einzug hält und auch im Alltag an Bedeutung gewinnt. Viele elektronische Verbraucher werden kleiner, handlicher, ja nahezu filigraner. Wenn man nur einmal die Entwicklung der Handytechnologie betrachtet, zeigt sich, dass Handys heute kleine Hochleistungscomputer mit leistungsstarken Prozessoren sind. Diese Entwicklung wurde nur möglich, da sich die Wissenschaft und die Softwareentwicklung auf die Mikrotechnologie eingestellt haben. Eine Vielzahl der heutigen Handys mit einem integrierten Kamerachip bis zu 13 Mio. Pixel (Stand 02.12. 2012) besitzt sogar eine Panoramafunktion. Dies ist nicht nur den immer kleiner werdenden Kamerachips zu verdanken, sondern auch der digitalen Unterstützung zur Bearbeitung der aufgenommenen Fotos. Angepasste Algorithmen der Bildverarbeitung an die neuen Systemarchitekturen von Laptops und Handys führen zu alltagstauglichen Bildbearbeitungen, wie dem Zusammenführen von mehreren Einzelaufnahmen zu einem flächendeckenden Panorama. Dieses Verfahren nennt man auch Stiching und wird mit Hilfe von Matching Algorithmen umgesetzt.

Auch im Bereich der ferngesteuerten Fluggeräte hat die Mikrotechnologie Einzug gehalten. Aufgrund dieses Fortschritts ist es heute möglich, Hochleistungsrechner und kleinste Komponenten wie GPS, Kompass und Neigungssensoren in Modellflugzeuge einzubauen. Somit lassen sich vorher definierte Flugrouten automatisch abfliegen. Dank der immer effizienteren Energienutzung können ferngesteuerte Fluggeräte heute schon mehrere Stunden ohne Zwischenlandung, im Militär auch mehrere Tage, in der Luft gesteuert werden.

Durch die Kombination von Bildverarbeitung und Mikrotechnologien im Modellflug können mittels UAS Geoinformationen aus der Luft erfasst werden. Basierend auf dieser Entwicklung wurden bereits Luftbildaufnahmen von UAS getätigt. Diese wurden zur Generierung von Orthophotos in der photogrammetrischen Auswertungssoftware LPS verwendet. Die Luftbilder der

Drohnen konnten durch die Kombination der aufgenommenen Flugparameter mithilfe des Programms jedoch nicht ausgewertet werden und führten somit nicht zum erwarteten Ergebnis.

Ziel meiner Arbeit ist es, aufbauend auf dieser Problemstellung mit Hilfe von Open Source Bibliotheken aus der *Computer Vision*, die Schwierigkeiten, die bei der benannten Auswertung entstanden sind, zu umgehen. Dazu werden im Folgenden genauere Ausführungen über Motivation, Zielsetzung sowie Aufbau, Methodik und Problemstellung der Arbeit getätigt.

1.1 Motivation

Die Auswertung von Geoinformationen aus der Luft soll zeitnah und möglichst selbständig erfolgen. Die Photogrammetrie ist eine bewährte Methode, Bildverbände zu georeferenzieren und daraus Koordinaten zu extrahieren.

Aktuelle Forschungen zeigen, dass sich der hohe Automatisierungsgrad der aus der konventionellen Photogrammetrie bekannten Arbeitsabläufe der Bildzuordnung, nicht ohne Weiteres auf die Auswertung von UAS-Bilddaten übertragen lässt. Hierzu gab es im September 2009 an der Eidgenössischen Technischen Hochschule Zürich (ETH) eine Konferenz (die „uav-g“) zur Nutzung von UAVs in der Geodatenerfassung unter der Leitung von Prof. Ingensand und Dr. Eisenbeiss vom Institute of Geodesy and Photogrammetry (IGP). Unter anderem wurde hier deutlich, dass bis dato keine Standard-Lösungen zur Auswertung solcher Luftbilder existieren. Hierfür sind Entwicklungen und Optimierungen von Methoden für die Prozessierung von Drohnen-Luftbildern notwendig. Seitdem beschäftigen sich eine Vielzahl von Universitäten und Hochschulen auf der ganzen Welt mit der Geoinformationsgewinnung aus Drohnenluftbildern.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist die Untersuchung von Methoden, mit denen eine automatisierte Verknüpfungspunktsuche von Bildern mit der Programmbibliothek OpenCV[1] erreicht werden. Hierzu werden praxisnah Drohnenluftbilder

[1] OpenCV ist eine Programmbibliothek und wird im Abschnitt 5 näher erklärt.

von einem Projekt der GEO-METRIK IG mbH[2] verwendet. Fokus dieser Arbeit ist das bekannte Korrespondenzproblem aus der Photogrammetrie. Vorhandene Softwares, welche bereits Lösungen für dieses Problem anbieten, werden vorgestellt. Beispielhaft werden einige Lösungsansätze mit Hilfe von OpenCV programmiert und deren Erfolg oder Misserfolg dokumentiert. Hierbei werden zudem Änderungen an den Ausgangsdaten vorgenommen. Dies alles geschieht unter dem experimentellen Aspekt und soll demnach keinen fertigen Workflow mit OpenCV vorweisen. Es soll gezeigt werden, welches Potenzial die OpenCV-Bibliotheken mit sich bringen und ob diese in der Praxis der Vermessung eine Verwendung finden können.

1.3 Aufbau und Methodik dieser Arbeit

Nach dem kurz der aktuelle Stand der UAV-gestützten Luftbildauswertung beschrieben wurde, werden im Anschluss daran Methoden zur automatischen Extraktion von Bildmerkmalen dargestellt und deren Zuordnungsverfahren in einem Bildverband aufgeführt. Danach wird die Programmbibliothek OpenCV aus der *Computer Vision* auf passende Algorithmen untersucht, welche zur Lösung der vorher beschriebenen Probleme beitragen. Eine Auswahl dieser Algorithmen wird anschließend in der Programmiersprache C++ an einem Fallbeispiel angewendet. Zum Schluss erfolgt ein Resumé, welches die Arbeit zusammenfasst und einen Ausblick darauf gibt, wie sich die UAV-gestützte Luftbildauswertung mit den Problemen der digitalen Luftbildauswertung in der Zukunft etablieren könnte.

Literaturrecherche insbesondere aus den Aufgabenbereichen Bildverarbeitung und Photogrammetrie gefolgt von einer experimentellen Untersuchung tragen zur Auswertung der Daten, die durch Befliegung mit UAV erhoben wurden, bei. Dabei wird versucht, aktuelle Forschungsstände der *Computer Vision* in die traditionellen Auswertemethoden der Photogrammetrie einzubeziehen. Dieses wird an einem Fallbeispiel aus der Praxis untersucht.

[2] GEO-METRIK Ingenieurgesellschaft mbH Stendal ist ein Vermessungsunternehmen aus Stendal und wird im Abschnitt 6 vorgestellt.

1.4 Das Korrespondenzproblem

Die Korrespondenzanalyse ist ein Verfahren zur automatisierten Bildzuordnung (*image matching*). Die korrekte Zuordnung gleicher Bildmerkmale ist eine der ältesten und zugleich aktuellsten Fragestellungen in der Photogrammetrie und des Computersehens (*Computer Vision*). [LUH03]

Die digitale Bildzuordnung von Luftbildern ist eine der grundlegenden Aufgaben in der Photogrammetrie und deren Automatisierung seit einigen Jahren Forschungsthema. Verfahren für eine erfolgreiche Bildzuordnung bei klassischen Luftbildauswertungen sind bereits schon länger in bekannten Programmen (LPS, INPHO, ImageStation) verfügbar. Diese Bildzuordnung besteht in der Bestimmung von Bildkoordinaten homologer Punkte von mindestens zwei Bildern. Im Vergleich zu herkömmlichen Luftbildern haben Drohnenluftbilder einige Eigenschaften, die eine Orientierung und Bildzuordnung erschweren. Durch eine vergleichsweise niedrige Flughöhe sind insbesondere im Gebirge extreme Höhenunterschiede festzustellen, die in einem Bild zum Teil ein Vielfaches der eigentlichen Flughöhe ausmachen können, da die Bodenabdeckung und damit die gefundenen vergleichbaren Objekte in einem Bild geringer sind als bei einem klassischen Bildflug aus großen Höhen. Auch sind die geplanten Flugstreifen nicht immer exakt parallel und auch die Drehwinkel der Drohnen PHI und KAPPA weisen Schwankungen auf (vgl. 6.1.2.). Aufgrund der geringen Nutzlast werden Kompaktkameras eingesetzt, bei denen man nur selten Kenntnis über die Kalibrierung besitzt.

Warum wird diese Problematik nochmals aufgegriffen? Kompaktkameras sollen es dem Benutzer so angenehm und leicht machen, scharfe und gut belichtete Bilder zu erstellen. Hierbei werden allerdings Automatismen verwendet, welche reproduzierbare Kamerakalibrierung unmöglich machen. Ohne die Parameter einer Kamerakalibrierung gilt es, die Problematik unter anderen Vorraussetzungen erneut zu erschließen.

Generell besteht das Problem darin, zwei identische Punkte zu identifizieren, welche nicht nur einheitliche Merkmale aufweisen, sondern wirklich zum gleichen, realen Objekt gehören. Allerdings sieht ein Objekt aus unterschiedlichen Kamerapositionen und dazu noch aus der Luft, nicht immer gleich aus und wird auf einem Bild auch immer nur zweidimensional dargestellt.

Dadurch ergeben sich für einen Bildpunkt P_{ij} (Punkt i im Bild j) nach LUHMANN [LUH10, S. 453] prinzipiell folgende Probleme:

- es existiert aufgrund von Verdeckung kein homologer Bildpunkt P_{ik}
- es existieren aufgrund von mehrdeutigen Objektstrukturen oder durchsichtigen Oberflächen mehrere potentielle Kandidaten P_{ik}
- Bildstörungen (Rauschen) können in texturarmen Regionen zu instabilen Lösungen führen.

LUHMANN [LUH10, S. 453ff] verdeutlicht zudem Vorraussetzungen, welche für praktikable Lösungen gegeben sein sollten. Diese können allerdings nicht immer auf Bildflüge mit Drohnen gewährleistet werden:

- Gängige Lösungen der Bildzuordnung gehen in der Regel davon aus, dass die Intensitätswerte aller benutzen Bilder im gleichen spektralen Bereich liegen. Davon kann auch bei Bildflügen mit Drohnen ausgegangen werden, da die Nutzlast hier eingeschränkt ist und meist nur mit einem Kameramodel geflogen wird.
- Die Beleuchtung, atmosphärische Einflüsse und Medienübergänge sollten während der Bildaufnahmen konstant sein.
- Objektoberflächen sollten undurchsichtig und formstabil sein und stückweise glatte Objektoberflächen besitzen.
- Bildüberlappungen sollten eingehalten werden und Näherungswerte der Orientierungen sollten bekannt sein. Diese beiden Parameter sind bei einem Drohnenflug mit den heute verbauten Komponenten nicht zu realisieren, weil GPS und IMU zu ungenau sind.

Damit eine Auswertung und Aufbereitung der Luftbilder effektiv gestaltet werden kann, benötigt diese Alternative der Photogrammetrie die Programmierung einer speziell auf den Drohnenflug angepassten Softwaretechnologie. Hierbei kommen neue Verfahren der digitalen Bildauswertung wie SURF und SIFT zum Einsatz. Anhand der aufgezeichneten Flugparameter der Drohne, der Luftbilder und der gemessenen Passpunkte, können die Luftbilder im Raum orientiert werden. Aufgenommene Luftbilder mit einer handelsüblichen Digitalkamera haben den Makel, dass aufgenommene Objekte la-

geversetzt erscheinen. Das hat zur Folge, dass höhere Objekte umliegende niedrigere Objekte auf einem einzelnen Foto verdecken. Dieser Effekt ist der Zentralperspektive geschuldet, bei der alle Projektionsstrahlen durch ein gemeinsames Projektionszentrum gehen und verursacht, dass Objekte liegend erscheinen. Solche Aufnahmen eignen sich nicht für vermessungstechnische Auswertungen. Die Lage der Objekte muss korrigiert werden. Die Experten sprechen hierbei von Entzerrung. Dafür wird das aufgenommene Gelände in einer senkrechten Parallelprojektion auf einer horizontalen Ebene wiedergegeben. Diese entzerrten Aufnahmen werden Orthobilder oder Orthoimages genannt. Um die Entzerrungen vornehmen zu können, werden allerdings Höheninformationen über die abgebildete Geländeoberfläche und deren Objekte benötigt. Diese Höheninformationen stammen aus vor Ort gemessenen Passpunkten. Durch das Fotografieren mit überlappenden Bildpaaren können die Objekte aus verschiedenen Blickwinkeln betrachtet werden. Dadurch lassen sich Höhenunterschiede zwischen dem Objekt und der Erdoberfläche berechnen.

Bei Flügen mit einem UAV zur Erstellung von Bildmosaiken oder sogar 3D-Modellen, gibt es einige Grundvoraussetzungen für die klassische Bildauswertung, die nicht erfüllt werden können. Dies liegt an den komplett unterschiedlichen Flugeigenschaften dieser beiden Verfahren.

1.5 UAVs in der Geodatengewinnung - Anwendungsszenarien

Mit dem Einzug neuer Technologien hat sich das Aufgabenfeld des Geodäten in den letzten Jahren erweitert. UAVs werden schon häufig in der Vermessung eingesetzt, da sie die Lücke zwischen der terrestrischen Vermessung und der klassischen Luftbildphotogrammetrie schließen.

Oftmals genügen einfache Fotos aus der Luft zur Dokumentation von einzelnen Bauphasen. So werden UAVs, ausgestattet mit Digitalkameras, für Bauwerksdokumentationen eingesetzt und deren Fotos als zusätzliche Informationen an Bestandpläne oder Bauwerkszeichnungen angefügt. Dadurch lassen sich einzelne Bauteile in großen Höhen fotografieren und auf Beschädigungen untersuchen. Die Nutzungen von UAVs zur Dokumentation von

Bauwerken während der Planungsphase, während der Bauphase und nach Fertigstellung sind möglich.

Abb. 1: Luftbild während der Bauphase der Elbbrücke bei Schönebeck (Quelle: GEO-METRIK IG mbH Magdeburg)

In Abb. 1 wurde der Baufortschritt während einzelner Bauphasen an der Elbbrücke bei Schönebeck dokumentiert. Somit können Informationen zu einem ganz bestimmten Zeitpunkt flächendeckend belegt werden. Die Luftbilder entstanden mit dem Quadrocopter[3] md4-1000 und wären aufgrund des finanziellen Aufwandes ohne den Einsatz eines UAV nicht möglich gewesen.

Weiter werden UAVs bei der Planung von Kabeltrassen eingesetzt. Einzelne Luftbilder lassen sich auf bestehende Lagepläne oder Grundbuchkarten entzerren und somit georeferenzieren. Wie in Abb. 2 zu sehen ist, werden hierbei Transformationsgenauigkeiten von wenigen Zentimetern erreicht. Die transformierten Bilder lassen sich gut mit Planungsunterlagen kombinieren und bieten so dem Planer eine aktuelle Übersicht.

[3] Ein Quadrocopter ist ein Fluggerät (UAV) mit vier Rotoren.

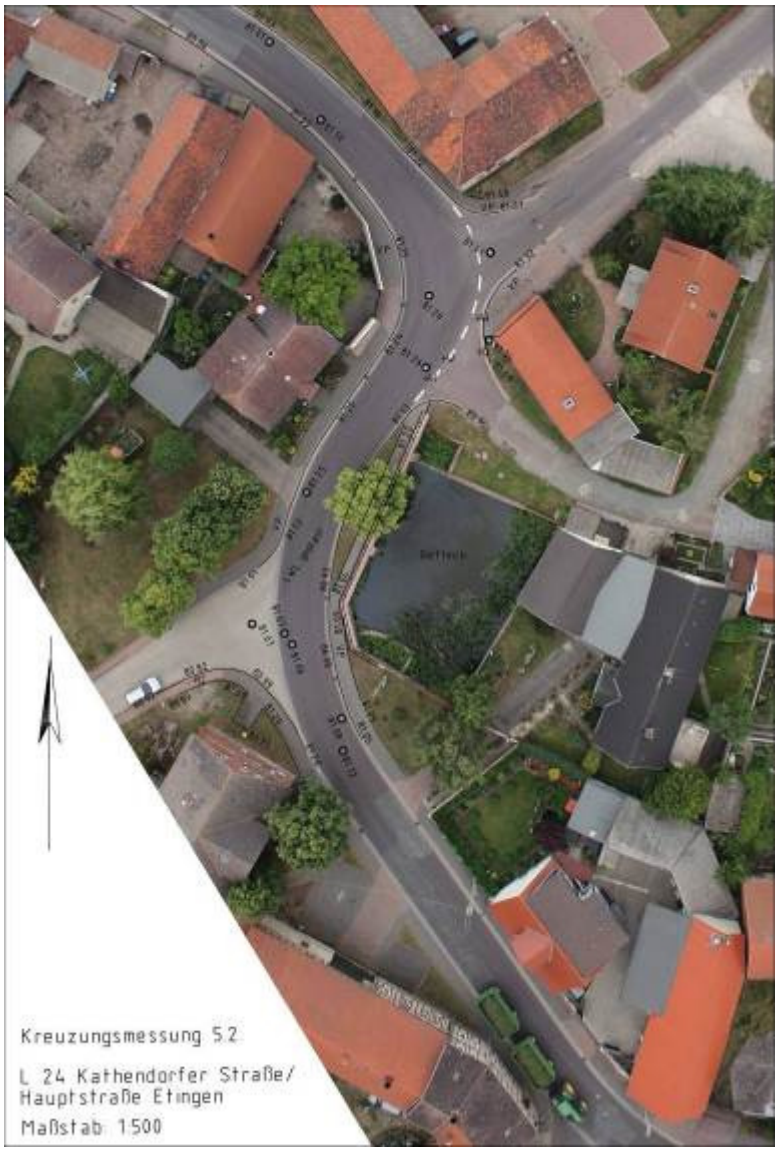


Abb. 2: Planungsunterlagen zum Neubau einer LWL-Trasse (Quelle: GEOMETRIK IG mbH Stendal)

Nun kann der Planer Maße aus dem Plan oder Foto abgreifen, ohne dass dieser vor Ort mit dem Bandmaß nochmals messen muss. Auch ist die zusätzliche Information der aktuellen Bilddaten bei Ämtern oder bei den Bewohnern ein Mehrgewinn, um die Planungen voranzutreiben oder zu diskutieren. Bei Umtrassierungen von Leitungen kann somit ganz genau geschaut werden, auf welchem Grundstück sich Leitungen befinden und ob eventuell gerodet oder die Leitungen mittels Vortrieb durch das Erdreich gebohrt werden müssen. Diese zusätzlichen Informationen haben auch einen immensen wirtschaftlichen Aspekt, da unnötige Kosten oder Fehlkalkulationen vermieden werden können.

Auch Volumenberechnungen von Erdmassen lassen sich auf Grundlage der Drohnenluftbilder erstellen. Durch die Kombination aus Drohnenbildern aus der Luft und terrestrisch aufgenommenen Passpunkten (GCP) am Boden, lässt sich eine georeferenzierte, dreidimensionale Punktwolke der Oberfläche erstellen. Eine solche Punktwolke ist ausreichend, um für Haufwerke oder Baustoffhalden ohne Bewuchs eine Masse zu ermitteln. Ist in dem zu berechnenden Gebiet Bewuchs vorhanden, muss dieser aus der Punktwolke eliminiert werden, da es sich hier um ein optisches Verfahren handelt. Durch diese Massenberechnungen können Baukosten überwacht oder zur Kostenabrechnung beim Bauherrn verwendet werden. Je nach Geländebeschaffenheit, Genauigkeit der Passpunkte, verwendetem UAV und verwendeter Kamera können so Massen ermittelt werden, welche eine Differenz von ca. 0,5 % gegenüber den terrestrisch ermittelten Massen beträgt. Bei den folgenden Abbildungen handelt es sich um ein Baustoffhaufwerk für eine Deichsanierung bei Wittenberge. In Abb. 3 ist ein digitales Oberflächenmodell (DOM) zu sehen, welches aus der Punktwolke erzeugt wurde. Abb. 4 zeigt eine dreidimensionale Ansicht des Haufwerks mit realer RGB-Flächenfärbung der Vermaschungen.

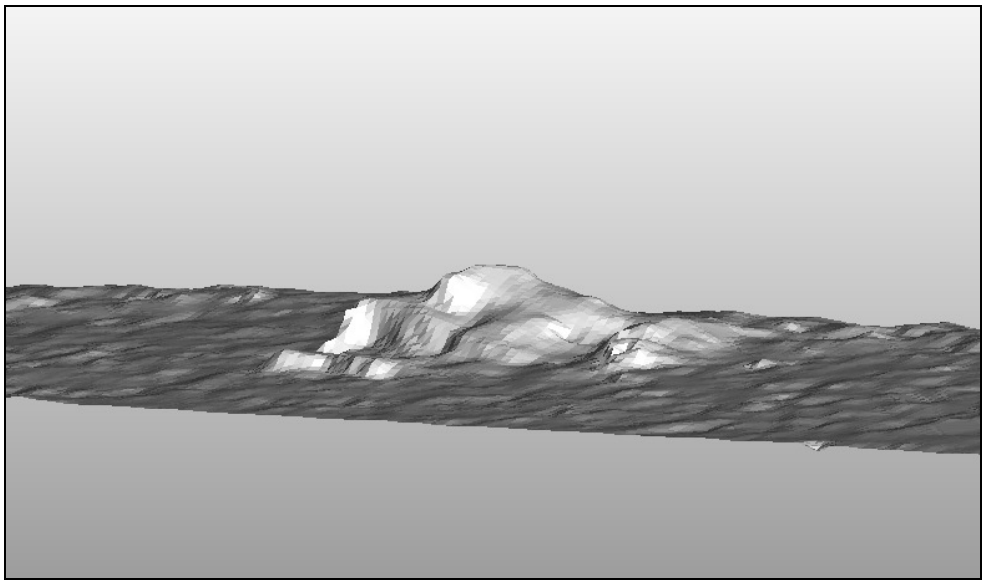


Abb. 3: DOM von einem Baustoffhaufwerk (Quelle: eigener Entwurf)

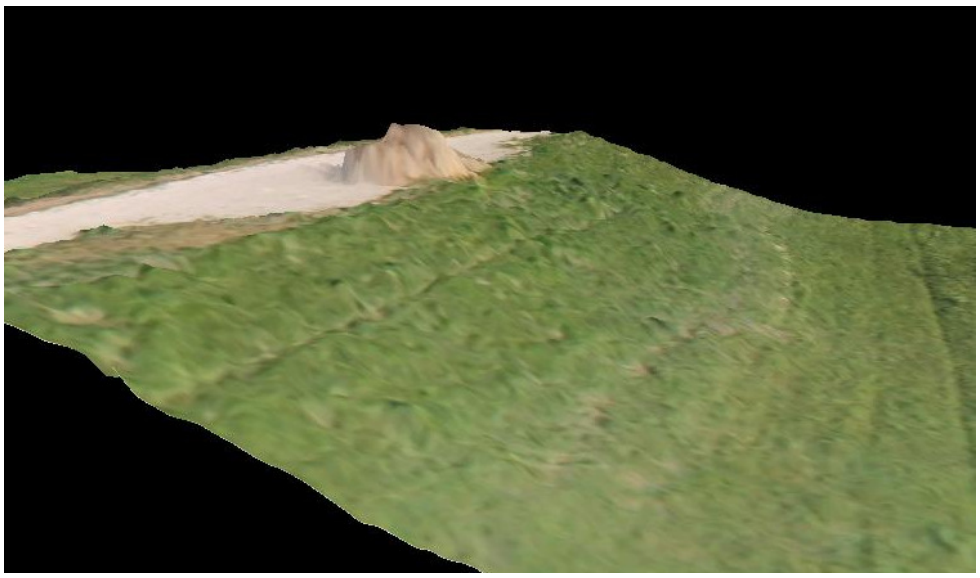


Abb. 4: 3D-Ansicht von einem Baustoffhaufwerk auf einem Deich bei Wittenberge (Quelle: eigener Entwurf)

In Tabelle 1 werden die unterschiedlichen Ergebnisse aus Befliegungsdaten und terrestrischer Vermessung aufgezeigt. Zudem werden die Aufnahme- und Bearbeitungszeiten gegenübergestellt.

	Befliegung mit md4-1000	terrestrische Messung		Differenzen in %
Aufnahmezeit	10 min	30 min		-67%
Bearbeitungszeit	2 h	1 h		100%
Anzahl Einzelpunkte	5'000	36		13'788%
ermittelte Massen	9.9 m³	9.8 m³		1%

Tabelle 1: Gegenüberstellung von Befliegung und terrestrischer Vermessung von einem Baustoffhaufwerk auf einem Deich bei Wittenberge (Quelle: eigener Entwurf)

Wenn man die reine Aufnahmezeit betrachtet, ist die Befliegung um die Hälfte der Zeit der terrestrischen Messung schneller. Viel deutlicher wird dieses Ergebnis mit steigender Flächengröße. In diesem Beispiel ist die aufgenommene Fläche relativ klein gewählt. Allerdings ist die Bearbeitungszeit im Innendienst bei der Befliegung deutlich höher. Das Doppelte an Zeit wird für die Auswertung der Befliegungsdaten gebraucht. Dieses Verhältnis bleibt auch bei steigender Flächengröße in etwa gleich. Was aber wirklich entscheidend für eine genaue Massenermittlung ist, ist die Anzahl der gemessenen Einzelpunkte. Denn jede Erhebung die nicht erfasst oder generalisiert wird, ist eine Verfälschung des Ergebnisses. Die Anzahl der Einzelpunkte

aus der Befliegung ist 138-mal größer als aus der terrestrischen Messung. Durch die hohe Anzahl der Einzelpunkte wird das Gelände viel realistischer dargestellt, womit die Berechnung der Masse auch realistischer wird. In diesem Beispiel spielt der Massenunterschied keine große Rolle. Diese Massenberechnung wurde nur zu Vergleichszwecken erstellt. Geodätisch interessant werden Befliegungen mittels Drohne dann, wenn das Gelände nicht begehbar ist, weil dies für Menschenleben gefährlich sein könnte oder die Fläche wirtschaftlich zu klein für eine Befliegung mittels Flugzeug ist. Ab wann allerdings ein UAV wirtschaftlicher ist, hängt ganz stark von dem UAV und dessen Flugzeit bzw. Reichweite ab. Bei dem hier eingesetzten Quadrocopter ist eine Befliegung ab einer Flächengröße von ca. 3 km² nicht mehr wirtschaftlich zu bearbeiten.

Je nach Art der Anwendung, können unterschiedliche UAVs verwendet werden. Welche verschiedenen Arten es von UAVs gibt, wird im nächsten Abschnitt aufgezeigt.

2. UAV-gestützte Luftbildauswertung

UAVs sind unbemannte Flugsysteme und wurden 1783 durch die Gebrüder Joseph Michel und Jacques Etienne Montgolfier erstmals eingesetzt. Montgolfier ließen damals einen Heißluftballon steigen [BRA07, S.70 ff]. Dieser ist mit den heute eingesetzten UAVs nicht mehr zu vergleichen, denn neben der militärischen Weiterentwicklung hielt die Mikrotechnologie auch in Bereichen des Modellbaus Einzug. Ferngesteuerte Modellflugzeuge existieren schon seit über 25 Jahren, nur sind die heutigen Modelle mit GPS (Global Positioning System) und INS (Inertiales Navigation System) ausgestattet und werden auch unbemannte autonom navigierende Flugsysteme genannt, oder kurz UAS (unmanned aircraft system). Zudem überwachen heute zu jeder Zeit des Fluges kleine Mikrosensoren auch den Akkuzustand und die Motorleistung eines Fluggerätes. Durch die Kombination aus Sensoren und Positionsinformationen aus dem GPS-Modul können vordefinierte Waypoint-Routen geflogen werden. Dank der Mikrotechnologie, der ständigen Weiterentwicklung der Akkus und der verwendeten Leichtbaumaterialien können die Flugsysteme immer länger in der Luft gehalten werden. Somit wurden in den letzten Jahren kleinere UAVs wie Multikopter[4] oder Starrflügler[5] auch für die Geodatenerfassung aus der Luft zunehmend interessanter. War auf der Intergeo[6] im Jahr 2009 noch kein einziger Aussteller, welcher sich mit UAV beschäftigt hat, waren 2010 schon zwei Aussteller, 2011 schon 8 und 2012 bereits über 20 Aussteller vertreten. Generell kann festgestellt werden, dass sich die internationale UAV-Branche rasant weiterentwickelt hat. Durch diese dynamischen Neuerungen werden UAVs heute schon in den verschiedensten Anwendungsbereichen wie im Militär, Precision Farming oder in der Archäologie erfolgreich eingesetzt.

Auch in der Geodatenerfassung finden UAVs immer mehr Verwendung. Anfangs wurden Heißluftballons, ausgestattet mit Kamerasystemen und GPS, als Verfahren genutzt, um unbemannte Luftbilder zu erstellen. Heute werden UAVs in jeglicher Ausführung für Luftbilder verwendet. Doch welche Anforde-

[4] Als Multikopter werden all die Systeme bezeichnet, welche mehr als zwei Propeller für den Antrieb nutzen.
[5] Starrflügler (engl. fixed wings) besitzen feststehende Flügel und nutzen ein bis zwei Propeller für den Antrieb.
[6] Intergeo ist die weltgrößte Messe für Geodatenerfassung und –verarbeitung (Link: http://www.intergeo.de).

rungen werden an ein UAV in der Geodatenerfassung gestellt? Die folgende *MindMap* soll die Anforderungen für ein perfektes UAV verdeutlichen.

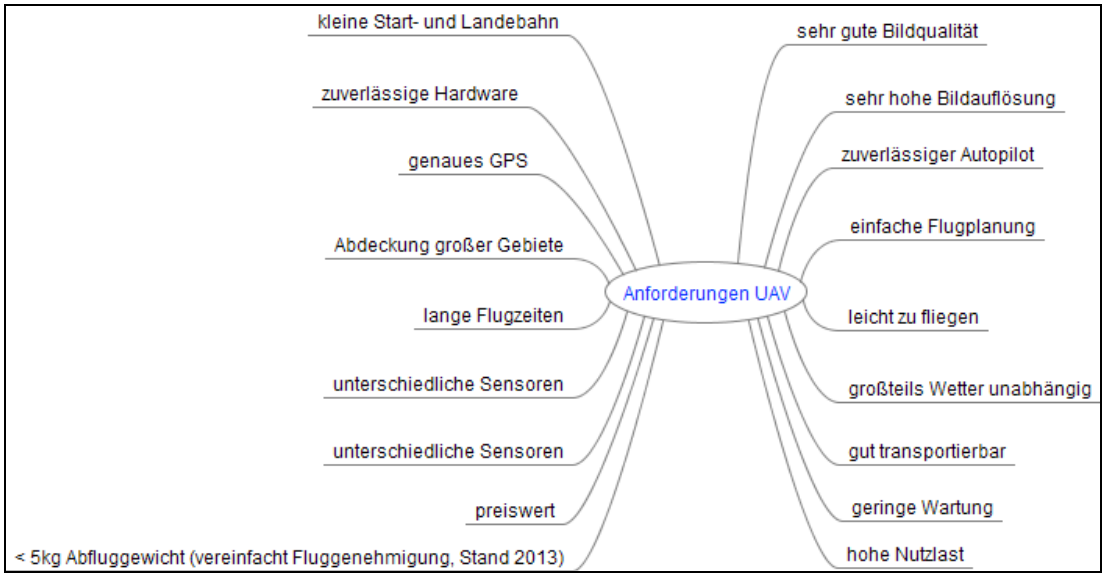
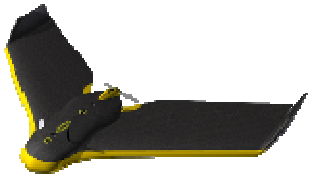


Abb. 5: MindMap für das perfekte Befliegungs- UAV (Quelle: eigener Entwurf)

In den nächsten Abschnitten wird der derzeitige Stand von UAVs und von Auswertesoftware für die Geodatenerfassung vorgestellt und kurz erläutert.

2.1 UAV-Typen

Es gibt viele verschiedene Typen von UAVs. Bei den zivilen UAVs unterscheidet man hauptsächlich zwischen Starrflügel-UAV, Hubschraubern und Multikoptern. Es gibt zudem auch Heißluft-Luftschiffe und mit Gleitschirm geflogene UAVs. Diese findet man in der Geodatenerfassung allerdings eher selten. Die folgende Tabelle 2 zeigt die unterschiedlichen Eigenschaften der drei häufigsten Systeme.

	Positiv	Negativ	Produktbeispiel
Starrflügel	+ selten Startbahnen benötigt + gute Sensoren + gute Autopiloten + hohe Fluggeschwindigkeit + Abdeckung großer Flächen + recht wetterunabhängig + fast lautlos	- braucht Landebahn - empfindlich bei Landungen - erfordert viel Flugerfahrung - schlechte Bildqualität durch hohe Fluggeschwindigkeit - sehr träge in der Luft - Planung der Flüge ist aufwendig - nur senkrechte Fotoaufnahmen möglich	• „eBee“ der Firma senseFly aus der Schweiz 
Hubschrauber	+ sehr geringe Start- und Landefläche (2-3m²) + meist hohe Zuladungen möglich + lange Flugzeiten bis 30 min + gute Autopiloten + präzise Flugmanöver möglich	- Pilot muss Experte sein - wartungsintensive Technik - oft mit Kraftstoff betrieben - sehr laut - sehr vibrationsanfällig - wartungsintensiv - schwer (>5kg) - Abdeckung kleiner Flächen	• „aeroscout“ von der Firma Aeroscout aus der Schweiz 
Multikopter	+ sehr geringe Start- und Landefläche (2-3m²) + sehr leicht (< 5kg) + gute Autopiloten + einfach zu bedienen	- geringe Nutzlast - oft geringe Flugzeiten - sehr vibrationsanfällig - Abdeckung kleiner Flächen - komplizierte Elektronik	• md4-1000“ von der Firma Microdrones aus Deutschland

	+ Bildaufnahmen aus allen Blickwinkeln möglich + sehr präzise steuerbar + sehr wendig + recht wetterunabhängig + kaum Fluggeräusche		

Tabelle 2: Eigenschaften der drei häufigsten UAV Typen (Quelle Tabelle: eigener Entwurf, Quelle Bilder: Internetseite der Hersteller)

Die maximalen Flugzeiten werden von allen UAV-Herstellern regelmäßig nach oben angepasst, da die Weiterentwicklungen der Akkus und Motoren ständig voranschreiten.

Für den Einsatz von UAVs bei Luftbildbefliegungen sind zwei Kriterien sehr entscheidend. Zum einen soll bei einem Flug möglichst viel Fläche abgedeckt werden können, um gleichbleibende Aufnahmebedingungen zu gewährleisten. Zum anderen soll die Zuladungskapazität an Foto- und Messausrüstung ausreichend sein. Das Diagramm in Abb. 6 zeigt die Einordnung der UAV-Typen in Bezug auf Durchschnittswerte von Zuladung und abdeckbarer Fläche.

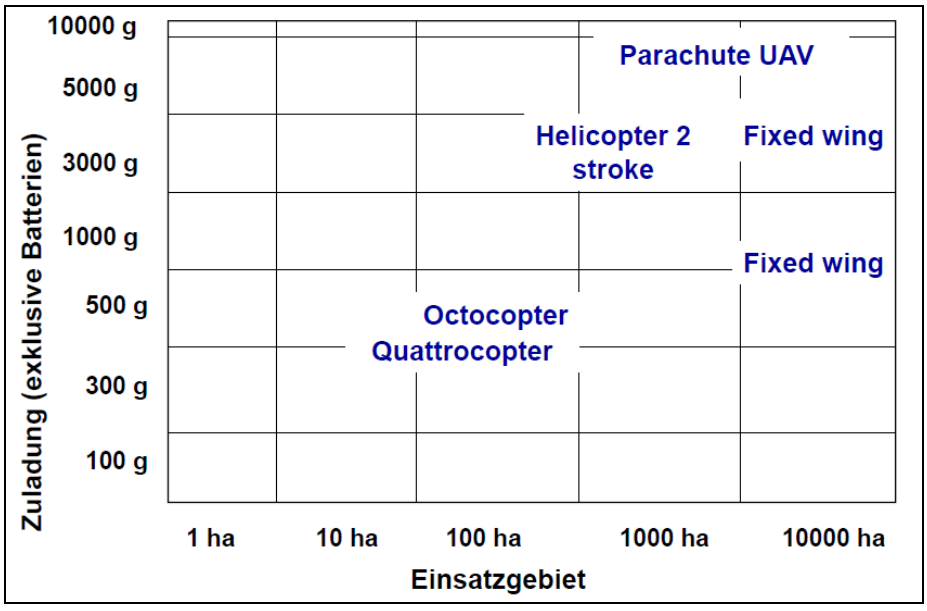


Abb. 6: UAVs - ihre Zuladung und Einsatzgebiet (Quelle: [THA, S. 19])

In Abb. 6 wird deutlich, dass ein weiteres UAV, sowohl in Nutzlast als auch in der Größe des Einsatzgebietes, ideale Voraussetzungen für den Einsatz in der Photogrammetrie hat, nämlich ein UAV mit Gleitschirm. Diese UAVs werden jedoch momentan nur sehr selten in Deutschland eingesetzt, da sie einige Nachteile vorweisen, welche den täglichen Einsatz erschweren: Gleitschirm-UAVs brauchen eine Start- und Landebahn von mindestens 25 m, wiegen weit über 5 kg und müssen gesondert bei Behörden angemeldet werden. Allerdings besitzen die Gleitschirm-UAVs eine derzeitige Nutzlast von bis zu 8 kg und können länger als eine Stunde in der Luft bleiben. Ein Beispiel für ein Gleitschirm UAV ist in Abb. 7 zu sehen.

Abb. 7: „SUSI 62“: ein Gleitschirm-UAS von der Firma Geo-Technic (Quelle: www.geo-uas.com)

Klar ist, dass der Einsatz eines UAV im konkreten Projekt immer im Einzelnen abgewogen werden muss. Es gibt nicht *das perfekte UAV* für die Geodatenerfassung, allerdings UAVs bei deren Einsatz je nach Aufgabenfeldern beispielsweise Wirtschaftlichkeit, Komfort oder Präzision von großem Vorteil sein können.

2.2 Verwendbare Sensoren

Im Gegensatz zur klassischen Befliegung mittels Flugzeug muss man bei den Sensoren eines UAVs deutliche Abstriche machen. Diese sind der geringeren Nutzlast der UAVs im Vergleich zum Flugzeug geschuldet. Nichtsdestotrotz lassen sich UAVs schon heute mit den verschiedensten Sensoren zur Datenerfassung bestücken. Abb. 8 zeigt Sensoren für die beiden

Quadrocopter md4-1000 und md4-200. Die Multispektralkamera wird im Bereich des Precision Farming und der Forstwirtschaft eingesetzt. Hiermit lassen sich falsche Düngung von Ackerflächen oder Waldschäden durch Baumkrankheiten erkennen. Die Taglichtkamera wird zur Echtzeit-Überwachung eingesetzt. Aufgrund ihrer geringen Masse kann das UAV länger in der Luft bleiben. Die Infrarotkamera kann bei Schadensdokumentationen und -erkennung von Solarfeldern eingesetzt werden. Durch die unterschiedliche Absorption des Sonnenlichts von korrekt funktionierenden gegenüber defekten Solarpanels lassen sich Schäden von Solaranlagen bei Tag gut erkennen. Die lichtempfindliche Schwarz-Weiß-Videokamera ist für Überwachungen während der Dämmerung gut geeignet. Diese wird auch von der niedersächsischen Polizei eingesetzt. Eine HD-Videokamera für die Videoaufnahme ist beispielsweise für Tierdokumentationen gut geeignet. Die Kompaktkamera kann hingegen von Landschaftsfotografen und Geodäten zugleich eingesetzt werden. Dies ist wohl der am häufigsten genutzte Sensor der md4-1000.

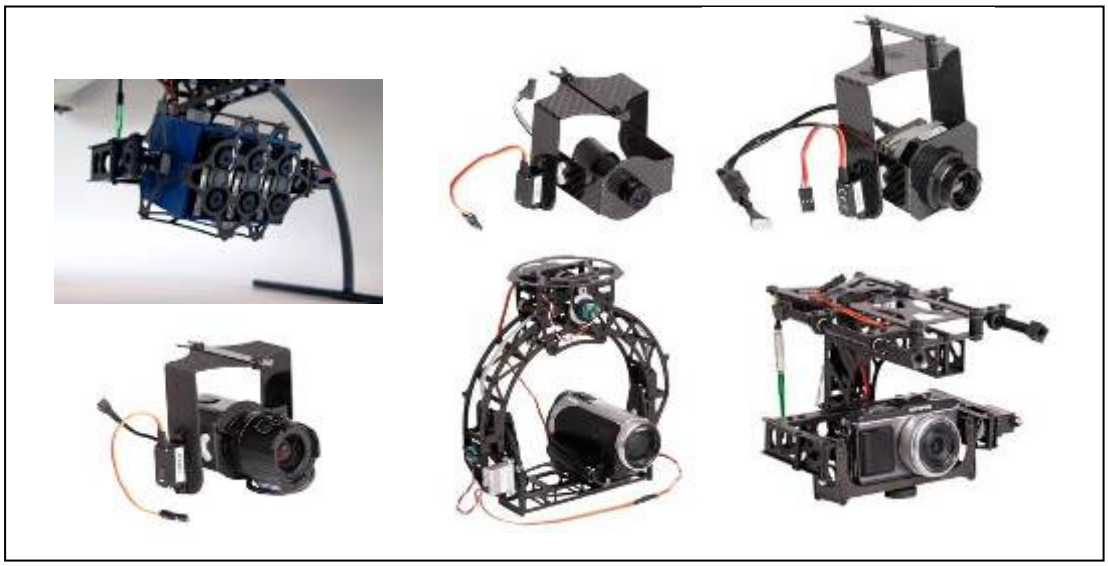


Abb. 8: Senoren für die Quadrocopter md4-1000 bzw. md4-200 (von oben links: 6 Kanal-Multispektralkamera, Taglichtkamera, Infrarotkamera; von unten links: lichtempfindliche Schwarz-Weiß-Videokamera, HD Videokamera, Kompaktkamera), (Quelle: www.microdrones.com)

Zu den hier aufgeführten Sensoren werden ebenso Sensoren in der Brandbekämpfung eingesetzt. Diese messen in den Rauchwolken, ob sich giftige

Dämpfe in der Luft befinden und ob eventuelle Evakuierungen vorgenommen werden müssen. Ein weiteres Einsatzgebiet zeigt sich derzeit in Tschernobyl, wo ein Geiger-Müller-Zählrohr in Kombination mit einem UAV getestet wird. In Fällen, wo es für Menschen gefährlich werden kann, bieten sich UAVs demnach als mögliches Werkzeug der Fernerkundung an.

Seit dem Frühjahr 2013 ist es ebenfalls möglich, einen Laserscanner mit der md4-1000 mitzuführen. Allerdings befindet sich die *microdrones GmbH* noch immer in der Testphase, da die Verbindung zur Bodenstation noch nicht korrekt funktioniert. Es bleibt jedoch abzuwarten, ob sich dieser Sensor in der Praxis bewähren wird, da die Genauigkeit der IMU[7]-Sensoren derzeit noch nicht ausreichend ist, um einen flächendeckenden dreidimensionalen Laserscan zu georeferenzieren. Gegenwärtig werden noch zusätzliche Passpunkte für die Georeferenzierung aufgemessen. Aber auch das zeigt eine stetig voranschreitende Entwicklung der Sensoren für UAVs.

2.3 Aktuelle Softwarelösungen für eine geodätische Auswertung

Ergebnisse einer geodätischen Auswertung können ein georeferenziertes Orthophoto und Mosaikbilder oder eine Punktwolke sein. Auf die Erzeugung dieser Produkte soll im Folgenden eingegangen werden.

Aufgrund ständig neuer Software-Veröffentlichungen werden die derzeit populärsten Softwarelösungen für die Auswertung von Drohnenluftbildern vorgestellt.

2.3.1 Bundler

Bundler ist eine Freie Software und wurde im August 2008 unter der GNU General Public License[8] das erste Mal veröffentlicht. Es kommt aus dem Bereich der Computer Vision und wurde von Noah Snavely von der Cornell University 2008 entwickelt und ist ein Softwarepaket, welches zur 3D-Rekonstruktion von Geometrien einer Bildszene genutzt wird. Hierbei liegt der Fokus nicht auf Drohnenbildern, sondern auf Bildern, welche willkürlich von Objekten jeglicher Art aufgenommen werden. Es arbeitet nach dem

[7] Die IMU ist eine Messeinheit, welche zur Bestimmung der Lage im Raum verwendet wird.
[8] General Public License berechtigt den Endnutzer die Software nutzen, studieren, verbreiten und ändern zu dürfen.

Structure-from-Motion (SfM) Verfahren und nutzt dafür unsortierte Bildsammlungen einer Objektszene. Bundler ist aus der früheren Software *Photo Tourism* entstanden [SNA06].

Diese 3D-Rekonstruktion ist in mehrere Schritte unterteilt. Mit Hilfe des SIFT-detectors werden in den Aufnahmen schrittweise korrespondierende Punkte gesucht und paarweise mit anderen Bildern verglichen. Basierend auf dem Levenberg-Marquardt (LM) Algorithmus werden anschließend mit Hilfe der allgemeinen Bündelausgleichung (BA) die Kamerapositionen aller Bilder rekonstruiert. Somit erhält man zu den Einzelaufnahmen die Parameter der inneren und äußeren Orientierung. Mit Hilfe der rekonstruierten äußeren Orientierung lässt sich nun eine grobe 3D-Punktwolke erzeugen. Diese Rekonstruktion einer Objektszene ist durch ihre geringe Verwendung von Parametern sehr rechenintensiv. Die flexible Bildanordnung, eine große Anzahl von Bildern sowie deren unterschiedliche perspektivische Verzerrungen stellen hohe Anforderungen an die automatische Auswertung. Je nach Größe des berechnenden Gebietes und Leistung des Rechners kann eine solche Berechnung mehrere Tage in Anspruch nehmen. Der Vorteil des Verfahrens liegt in der Möglichkeit Fotos eines Objektes mit einer oder verschiedenen unkalibrierten Kameras in relativ unsystematischer Aufnahmekonfiguration zu erzeugen und daraus automatisch ein dreidimensionales Modell des Objektes berechnen zu lassen. Da hier mit unkalibrierten Kameras gearbeitet wird und dieses Softwarepaket keine Einbindung von Kontrollpunkten (GCPs) vorsieht, ist die berechnete Punktwolke unmaßstäblich. Diese Punktwolke kann aber durch Weiterverarbeitung verdichtet und in ein globales Koordinatensystem transformiert werden [SNA06].

Mit Hilfe von Bundler entsteht nur eine dreidimensionale Punktwolke mit den RGB-Farbwerten aus den Fotos, aber kein flächendeckendes Mosaikbild oder Orthophoto.

2.3.2 Pix4D

Pix4D ist ein Softwarehersteller aus der Schweiz, der sich auf die photogrammetrische Prozessierung von Dohnenbildern spezialisiert hat. Er bietet einen Online-Cloud-Service und eine Desktop Software an, welche Pix4UAV

heißt. Diese ist eine Proprietäre Software, welche von Dr. Christoph Strecha entwickelt wurde. Zur Nutzung der Software kann man eine Desktoplizenz erwerben oder man bezahlt für den Online-Cloud-Service eine Gebühr für jedes umgesetzte Projekt. Bei beiden Systemen laufen die gleichen Prozesse zur automatisierten Auswertung ab. Es werden Fotos in die Software eingelesen. Anschließend wird eine ASCII-Datei mit den Parametern der Bildstandpunkte eingelesen. Es besteht nun die Option Passpunkte mittels ASCII-Datei einzulesen und diese in den Bildern zu identifizieren. Anschließend wird der Prozess mittels *Upload* für den Online-Cloud-Service und *Prozess* für die Desktopversion gestartet.

Ein Vorteil des Online-Cloud-Service ist die Verteilung der Berechnung auf mehrere Rechner. Dadurch kann ein Projekt schneller prozessiert werden. Vorher müssen allerdings alle Fotos auf den Server von Pix4D geladen werden, was bei schlecht ausgebauten DSL-Netzen ein großer Nachteil sein kann. In diesem Fall muss man auf die Desktop-Version zurückgreifen. Diese beansprucht allerdings Prozessor und Arbeitsspeicher stark, was das Weiterarbeiten am selben Computer extrem einschränkt. Zudem ist die Desktop-Version mit 15.000 € (Stand Feb. 2013) recht teuer. Beim Online-Cloud-Service wird nach der Größe des zu bearbeitenden Gebietes abgerechnet. Hier liegt der Preis für einen Quadratkilometer bei 240 € (Stand Feb. 2013), wobei man sich aussuchen kann, welche Produkte von einem Projekt benötigt werden. Bei Verwendung eines Bildmosaiks von einem Bildverband entstehen geringere Kosten.

Als Ergebnis erhält man ein Orthophoto, eine Punktwolke und ein Protokoll der Auswertung. Die einzelnen Arbeitsschritte werden betitelt, aber genauere Angaben zu den Arbeitsschritten während der Auswertung sind kaum bekannt, da sich die Entwickler sehr bedeckt halten. Die einzige Information für diese Arbeit ist, dass für die Verknüpfungspunktsuche Algorithmen verwendet werden, welche dem SIFT-Algorithmus sehr nahe kommen. [STR11, S. 2]

2.3.3 Agisoft Photoscan

Agisoft ist eine Proprietäre 3D Rekonstruktionssoftware, welche automatisch texturierte 3D-Modelle mittels Fotos eines Objektes erstellt. Agisoft LLC wur-

de 2006 in St. Petersburg gegründet und stützt sich auf die Berechnungsmethoden der Computer Vision. Die 3D-Rekonstruktion wird auch zur Erstellung von Geländemodellen verwendet, nur wird hier noch die Transformation der Punktwolke vorgenommen. Diese Software ist ausschließlich für den Desktopgebrauch konstruiert. Für die Rekonstruktion werden Fotos in die Software eingelesen. Anschließend kann man eine ASCII-Datei mit den Parametern der Bildstandpunkte und Passpunkte einlesen. Vor den einzelnen Berechnungsschritten ist eine Auswahl zwischen einigen Qualitätsstufen möglich. Je höher die Qualität der Berechnung sein soll, desto länger dauert die Auswertung. Als Ergebnis erhält man ein Orthophoto, eine Punktwolke und ein Protokoll der Auswertung.

Details zu den einzelnen Berechnungen sind nicht bekannt.

2.3.4 Zusammenfassung

Die Vorteile der Freien Software liegen in der detaillierten Beschreibung des Verfahrens. Die Proprietäre Software ist meist eine Blackbox, in die man Daten einliest, die Software diese verwertet und ein Ergebnis liefert.

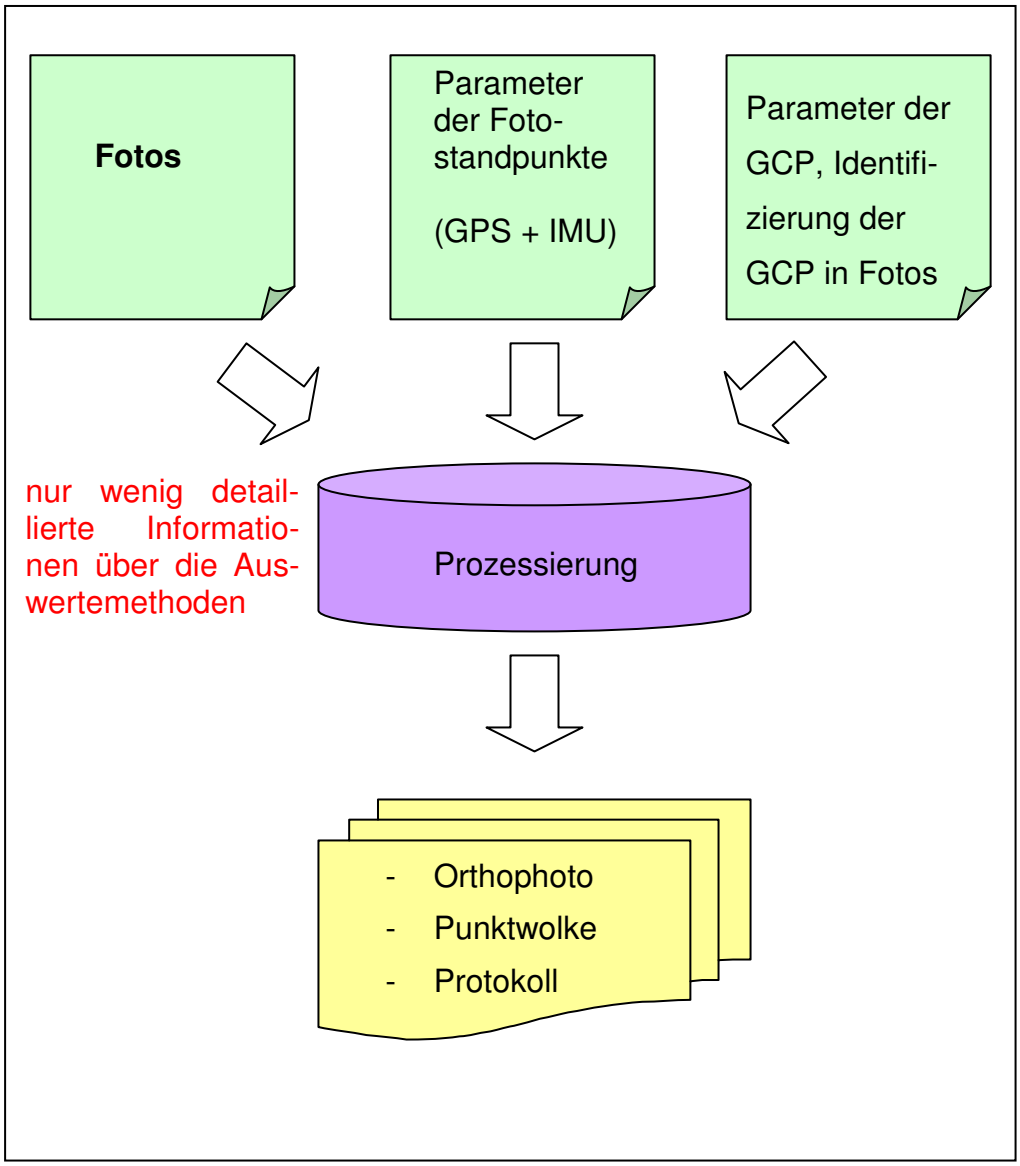


Abb. 9: Funktionsprinzip Proprietärer Software zur Bildprozessierung (Quelle: eigener Entwurf)

2.4 Vor- und Nachteile in der Praxis

Traditionelle photogrammetrische Luftbildaufnahmen aus Flugzeugen und Hubschraubern benötigen einen Flugplatz in Reichweite und Start- und Landegenehmigungen. Dies hat zur Folge, dass Planungsphasen und Reaktionszeiten sehr langwierig sein können. Ein UAV hat den Vorteil, dass es bis 5 kg Abflugmasse in Deutschland eine allgemeine Aufstiegserlaubnis bekommt. Diese ermöglicht es, außerhalb von Ortschaften ohne weitere Anmeldungen bis zu einer Flughöhe von ca. 100 m über dem Grund fliegen zu können. Die gesetzlichen Grundlagen sind in Deutschland auf Landesebene

geregelt und sind daher länderabhängig zu erfragen und zu beantragen. Durch die allgemeine Aufstiegserlaubnis lassen sich sehr schnell und kurzfristig Flächen befliegen. Zudem macht das Wetter eine langfristige Planung für einen klassischen Bildflug sehr schwierig, wobei man mit dem UAV einfach eine gute Wetterlage am Einsatzort abwarten kann.

Flugzeuge und Hubschrauber müssen eine gewisse Mindestflughöhe einhalten. In Deutschland beträgt diese über besiedeltem Gebiet 1000 Fuß (ca. 300 m). Dadurch ist deren Bildauflösung niedriger als der des UAVs, welches in einer Flughöhe von ca. 100 m fliegt und je nach Kamerasystem eine Bodenauflösung von ca. 3cm/Pixel aufweist. Man kann mit einer solchen Bildauflösung beispielsweise Risse in Straßen oder in Bauwerken erkennen und lokalisieren. Innerhalb weniger Stunden lassen sich somit aussagekräftige Schadensdokumentationen erstellen. In havarierten Gebieten ist somit ein schnelleres Eingreifen möglich, ohne dass sich Personen der Gefahrenstelle nähern müssen. Dieser Vorteil bewährt sich derzeit bereits in der Praxis. In der Zeit des Hochwassers Ende Mai und Anfang Juni 2013 traten zahlreiche Flüsse über ihre Ufer und überschwemmten große Teile besiedelter Flächen in Österreich, der Schweiz, Tschechien und Deutschland. Im Bereich der Saale und Elbe in Sachsen-Anhalt waren zwei UAVs von der GEO-METRIK AG für den Hochwasserschutz im Dauereinsatz. Mit ihren hochauflösenden Kameras und ihrer Reichweite von bis zu 1 km konnten sie Deichabschnitte befliegen und fotografieren. Diese Fotos wurden zur Kontrolle und Dokumentation der Schäden verwendet, ohne dass sich ein Mensch der Schadstellen nähern musste. Dadurch wurde zudem der Deich nicht zusätzlich belastet und es konnte trotzdem eine Sichtprüfung der Schäden erfolgen. Durch die photogrammetrische Auswertung der Deichbefliegungen lassen sich die Schäden der Deiche bemessen und protokollieren. Diese Befliegungen wurden ebenfalls mit Flugzeugen und Hubschraubern getätigt, nur ist zum einen die Auflösung der Kameras nicht so genau, um die Schäden zu dokumentieren. Zum anderen war für mehrere Wochen der gesamte Luftraum über der Elbe in Sachsen-Anhalt unter militärischer Überwachung. So durften nur Flugzeuge mit einer Sondergenehmigung fliegen und das auch nur über 500 m Höhe, um den Einsatz von Militär und Polizei nicht zu stören. Dieses Problem besteht mit den UAVs nicht, da sie in wenigen Sekunden gefährliche

Luftbereiche verlassen können und innerhalb weniger Minuten gelandet sind. Eine rechtliche Einschränkung gab es für UAVs in der Zeit des Hochwassers nicht.

Mit Hilfe der benutzen INS wird die Bedienung der UAVs nicht nur einfacher, sondern auch sicherer, planbarer und genauer. Dies hat zur Folge, dass ein großes Potenzial von UAVs in Kombination mit handelsüblichen Digitalkameras auch zunehmend in der photogrammetrischen Auswertung erwartet wird. Dies stellt, sowohl wissenschaftlich als auch wirtschaftlich, eine interessante Alternative zu einer Befliegung mit großformatigen professionellen Luftbildkameras dar. Die heutigen Forschungs- und Entwicklungsarbeiten zielen in Richtung einer vollständig automatisierten Auswertung von Luftbildflügen mittels UAV ab.

3. Die Verknüpfungspunktsuche

Verknüpfungspunkte sind eindeutig bestimmte homologe Punkte im Überlappungsbereich eines Bildverbandes. Diese können sowohl rechnergestützt als auch manuell identifiziert werden. Allerdings erhält man nach einem Bildflug mittels UAS, abhängig vom verwendeten System, eine Vielzahl von Bildern. Beispielsweise produziert man bei einem Bildflug mit einem Quadrocopter aus einer Höhe von 100 m und mit 80 % Überlappung bei einer Aufnahmefläche von einem halben Quadratkilometer ca. 175 Bilder[9]. Bei dieser Vielzahl an Bildern, kann man nicht fehlerfrei manuelle Verknüpfungspunkte identifizieren, erst recht nicht, wenn es sich um texturarme Untersuchungsgebiete handelt wie z.B. Wiesen oder Ackerflächen. Hier müssen automatisch suchende Algorithmen das manuelle Finden von zusammengehörigen Punkten übernehmen. Hierbei erhält man keine globalen Koordinaten, sondern eine Matrix, an welcher Stelle in welchem Bild sich der Verknüpfungspunkt befindet.

Die Schwierigkeit bei der Suche in Drohnenluftbildern liegt an der geringen Bodenabdeckung eines einzigen Bildes. Je nach Einsatzgebiet des UAS werden Bilder aus Flughöhen zwischen 50 m und 200 m gemacht. Je nach Kamerasystem und Objektiv hat man oftmals eine Bodenabdeckung unter 300 m x 300 m je Einzelbild. Im Vergleich dazu hat ein Bild aus der klassischen Befliegung je nach Flughöhe und verwendetem Messbildkamera ein Vielfaches an Bodenabdeckung. Dabei ist es einfacher homologe Punkte zwischen den Einzelbildern zu identifizieren. Durch die Kalibrierung der Messkameras und der genaueren Erfassung der Flugparameter lassen sich die Orientierungen der Bilder zueinander im Vorfeld berechnen. Somit etablieren sich bei der klassischen Photogrammetrie pixelbasierte Verfahren, da eine Nährungskoordinate des vermutlich gleichen Pixels im nächsten Bild berechnet wird. Pixelbasierte Verfahren vergleichen durch Überlagern der Bilder einzelne Pixel auf Korrespondenzen. Bei dem Merkmalsbasiertem Verfahren werden zunächst markante Punkte in einem Bild identifiziert und anschließend mit den markanten Punkten eines weiteren Bildes verglichen.

[9] Berechnet wurde dies mit den Parametern der Olympus EP-2 mit einem 17mm Weitwinkelobjektiv, welches standardmäßig mit dem Quadrocopter md4-1000 von Microdrones ausgeliefert wird. (Stand 02.02.2012)

Die hier vorgestellte Suche nach homologen Punkten ist das Merkmalsbasierte Verfahren. Dieses Verfahren ist in den folgenden vier Etappen untergliedert.

3.1 Feature extraction

Als Erstes wird die Identifizierung von interessanten Merkmalen sogenannter *feature points* vorgenommen. Je nach Untersuchungsgebiet und Bildauflösung können sehr viele *feature points* identifiziert werden. Verfahren zur Bestimmung solcher *feature points* werden im Abschnitt 4 genauer erklärt.

3.2 Feature description

Im zweiten Schritt bekommt jeder *feature point* eine Beschreibung seiner Merkmale und seiner Umgebung. Auch hier gibt es unterschiedliche Verfahren zur Beschreibung eines Punktes. Diese Algorithmen nennt man *feature descriptors.*

3.3 Image matching

Der dritte Schritt ist das *image matching.* Hier werden die *feature descriptors* aus mindestens zwei verschiedenen Bildern miteinander verglichen. Hierfür gibt es unterschiedliche Verfahren, um einen Vergleich der Merkmalsbeschreibungen vorzunehmen. Die wohl bekannteste Methode, um eine Vielzahl von Variablen, Vektoren oder Matrizen in der Programmierung zu vergleichen, ist die Brute-Force-Methode. Hierbei werden einfach wahllos Argumente miteinander verglichen bis es einen positiven Treffer gibt. Diese Methode kann unter Umständen sehr lange dauern. Auch hier gibt es weitere Methoden, die z.B. basierend auf Vermutungen vergleichen.

3.4 Zuordnungsverfahren

Im vierten Schritt sind es Zuordnungsverfahren, welche die Korrektheit der übereinstimmenden Bildpunktbeschreibungen überprüfen sollen. Wenn sich bei Bildaufnahmen bewegte Objekte in dem Aufnahmebereich befinden, sollen diese möglichst in diesem letzten Schritt als „falsch“ eingestuft werden.

Dies ist nicht nur bei bewegten Objekten der Fall, sondern auch bei Oberflächen, die schwache Konturen aufweisen.

Nach LUHMANN [LUH10, S. 453ff] werden hier Grundprinzipien einiger Zuordnungsverfahren genannt.

3.4.1 Relaxation

Relaxation ist ein Verfahren, welches die Abstandsmaße zu benachbarten Eigenschaften vergleicht. Stimmen die Abstandsmaße bei dem wahrscheinlich korrespondieren Punkt nicht mehr, wird entweder das nicht korrekte Maß aus dem Prozess entfernt und noch einmal durchlaufen oder das wahrscheinlich korrespondierende Verhältnis aufgelöst.

3.4.2 Zuordnung in Bildpyramiden

Hierbei werden die Zuordnungen der Merkmale in verschiedenen Auflösungsstufen verglichen. Die Zuordnungsergebnisse aus der groben Auflösungsstufe werden mit in die nächst feinere Pyramidenebene übernommen und mit zusätzlichen Merkmalen kombiniert. „Dieses Verfahren kombiniert Robustheit (grobe Merkmalserkennung) mit Präzision (Feinzuordnung in der höchsten Auflösung)“ [LUH10, S. 463].

3.4.3 Zuordnung mit RANSAC

Random Sample Consensus, kurz RANSAC, beurteilt die möglichen Punktzuordnungen anhand ihrer relativen Orientierung. Hierbei wird aus einer zufälligen Stichprobe von Merkmalen ein Modell berechnet und dieses mit den Punktzuordnungen verglichen. Passen Punkte nicht zu diesem Modell, werden sie verworfen. Wenn das Modell passt, werden die Punktzuordnungen als „korrekt“ anerkannt, bis ein Schwellwert an Treffern erreicht ist.[FIS81, S.381 ff.]

4. Feature Extraction

In diesem Abschnitt werden verschiedene Algorithmen vorgestellt, welche zur Berechnung von *local features* eingesetzt werden.

4.1 Was sind *local features*?

Ein *local feature*e ist ein lokales Merkmal in einem Muster eines Bildausschnittes, welches sich von den Mustern seiner unmittelbaren Umgebung unterscheidet. Nicht immer sind solche Muster auf den ersten Blick zu erkennen. Sie werden meist nach Änderung der Bildeigenschaften sichtbar. Bei diesen wird besonders auf Intensität, Farbe und Textur geachtet. [MIK08, S.178 ff.]

Des Weiteren können *local features* als Punkte, Ecken oder auch als Flecken in einem Bild identifiziert werden. Diese Merkmale bezeichnet man als „lokal“, da sie vorerst nur in einem bestimmten Bereich eines einzigen Bildes als Merkmal erkannt werden.

Ideale *local features* besitzen nach MIKOLAJCZYK [MIK08, S.182 ff.] folgende Eigenschaften:

- *Repeatability*: Dies ist das Maß des wiederholten Auffindens eines *features*. In zwei Bildern der gleichen Szene, aufgenommen von unterschiedlichen Standpunkten, wird von den sichtbaren Merkmalen ein hoher Prozentsatz in beiden Bildern gefunden. Da die Suche nach *features* erst in dem einen Bild und dann in einem zweiten Bild erfolgt, wird hier die Häufigkeit der Wiederholungen bestimmt. Das Identifizieren eines gleichen *features* soll somit unabhängig vom Blickwinkel der Aufnahme sein.

  Dieses Maß ist eines der wichtigsten Kriterien eines idealen *features* und kann nach MIKOLAJCZYK (vgl. ebd.) auf zwei unterschiedlichen Wegen erreicht werden: entweder durch Beständigkeit oder durch Robustheit.

  - *Invariance*: Die Beständigkeit der wiederholenden *feature*-Erkennungen kann mit der Immunität gegen äußere Einflüsse beschrieben werden. Sind in den Fotos große Abbildungsfehler zu erwarten, ist der bevorzugte Ansatz diese mathematisch zu modellieren und somit geeignete Methoden zur *feature*-

Erkennung zu entwickeln, die nicht von diesen mathematischen Transformationen betroffen sind.

- *Robustness*: Bei relativ kleinen Abbildungsfehlern genügt es, die Methoden der *feature*-Erkennung weniger empfindlich gegenüber Abbildungsfehlern zu machen, die Genauigkeit für die Erkennung von *features* wird verringert. Typische Abbildungsfehler, welche die Robustheit beeinträchtigen können, sind beispielsweise Bild-Rauschen, Kompressionsartefakte, Unschärfe.

- *Distinctiveness*: Eine klare Unterscheidung der einzelnen *features* untereinander verhindert falsche Zuordnungen. Die Intensitäten der Muster, die einem *local feature* zugrunde liegen, können in mehreren Bildern durchaus variieren. Trotzdem sollten ähnliche Muster unterschieden und gleiche identifiziert werden können.

- *Locality*: Hierbei geht es um die Lage von Objekten. Da es bei Bildern uneingeschränkt um zweidimensionale Abbildungen geht, können Objekte, die im Vordergrund sind, andere überdecken. Da die *features* lokal in einem Bild beschrieben werden, können die Probleme von Überdeckungen reduziert werden und erlauben somit eine einfache Modellannäherung an die geometrischen und photometrischen Abbildungsfehler zwischen zwei Bildern aus unterschiedlichen Aufnahmestandpunkten.

- *Quantity*: Die Anzahl der gefundenen *features* sollte hinreichend groß sein, sodass eine verwendbare Anzahl von *features* sogar an kleinen Objekten gefunden wird. Wie immer ist die optimale Anzahl von *features* abhängig von der Abbildung. Idealerweise sollte die Zahl der gefunden *features* über einen großen Bereich, intuitiv durch einen einfachen Schwellenwert, anpassbar sein. Die Dichte der *features* sollte den Informationsinhalt des Bildes reflektieren.

- *Accuracy*: Die Genauigkeit lässt sich nicht immer klar abgrenzen und kann in der Praxis auch unterschiedliche Definitionen aufweisen. Hier bezieht sich die Genauigkeit auf die Lageabweichung zweier gleichen *features* in zwei unterschiedlichen Abbildungen. Die *features* sollten in

beiden Bildern genauestens zueinander passen, ohne sich vom Maßstab oder möglichen Abbildungsfehlern beeinflussen zu lassen.

- *Efficiency*: Hiermit ist die Zeitdauer gemeint, die ein Programm für die Suche der *local features* in einem neuen Bild benötigt.

Wie lassen sich diese Eigenschaften bewerten? Jede dieser Eigenschaften ist von unterschiedlichen Einflüssen abhängig – seien es Abbildungsfehler oder die Rechenleistung des Computers mit dem die Auswertungen vorgenommen werden. Hierbei kommt es immer auf das Ziel der Auswertung an. Ist eine schnelle Auswertung erfordert, wird die Fehlertoleranz erhört und die Exaktheit der Identifizierungen sinkt somit. Solche Wichtungen können über die Einstellungen in den Programmen zur Suche der *features* vorgenommen werden. Bei der Änderung von Einstellungsparametern geht man eventuell bei einer anderen Eigenschaft Kompromisse ein. Diese Beurteilungen und Wichtungen der Eigenschaften eines idealen *features* sind immer von den auszuwertenden Abbildungsszenarien abhängig.

Repeatability ist in jeder auszuwertenden Szene bedeutsam und direkt abhängig von den anderen Eigenschaften. Je nach Anwendung können beispielsweise *invariance*, *robustnees* oder *quantity* ansteigen oder fallen.

Distinctiveness und *locality* sind konkurrierende Eigenschaften und können nicht gleichzeitig erfüllt werden: Je lokaler ein *feature* ist, desto weniger Informationen sind über das zugrunde liegende Muster verfügbar und umso schwieriger wird es, die *features* ordnungsgemäß zuzuordnen. Auf der anderen Seite werden im Falle der Bild-Mosaik-Anwendungen durch Verbindung planarer Objekte oder einfache Rotationen der Kamera Bilder durch eine globale *homography* verbunden. Hierbei gibt es keine Probleme mit Überdeckungen oder Tiefen-Diskontinuitäten. Unter diesen Bedingungen kann die Größe der lokalen Funktionen ohne Probleme erhöht werden. Dadurch ergibt sich eine höhere *distinctiveness*.

Ein erhöhtes Maß an *invariance* führt in der Regel zu einer reduzierten *distinctiveness*. Ähnlich verhält es sich bei *robustness* gegenüber *distinctiveness*. So werden einige Informationen ignoriert und einfach als Verzerrung betrachtet, um Stabilität in der Auswertung zu erreichen. Daher ist es wichtig,

eine klare Vorstellung über das erforderliche Maß an *invariance* oder *robustness* für eine bestimmte Anwendung zu haben. Es ist schwer, gleichzeitig eine hohe *invariance* und *robustness* zu erreichen. Zudem kann sich eine *invariance*, die nicht an die Anwendung angepasst ist, möglicherweise negativ auf die Ergebnisse auswirken.

Accuracy ist besonders dort wichtig, wo die Untersuchung auf Korrespondenz präzise sein muss, z.B. bei der Schätzung der Epipolargeometrie oder zur Kamerakalibrierung. Hier bewegt man sich im Bereich von Subpixel-Genauigkeiten.

Quantity ist besonders nützlich für Methoden der Szenenerkennung, wo es entscheidend ist, das Objekt des Interesses dicht abzudecken. Andererseits hat eine hohe Anzahl von *features* in den meisten Fällen auch negative Auswirkungen auf die Rechenzeit.

In der Literatur werden viele verschiedene Möglichkeiten der Identifikation von lokalen *features* diskutiert. Einige Studien zeigen das Finden von Ecken über die Krümmung von Konturen. Andere analysieren direkt die Bild-Intensitäten, z.B. basierend auf Derivaten oder Regionen mit hoher Varianz. Eine andere Forschungsrichtung wurde vom menschlichen Sehen inspiriert und verfolgt gezielt die ablaufenden Prozesse im menschlichen Gehirn. Seit einigen Jahren offenbart sich ein Trend zur *feature*-Erkennung, welcher sich als invariant gegenüber verschiedenen geometrischen Transformationen, einschließlich der Maßstabsänderung und perspektivischen Verzerrung, bewährt hat.

Diese speziellen Algorithmen zur Identifizierung von lokalen Merkmalen, nennt man *detector*s oder Detektoren. Da der Identifizierung von lokalen Merkmalen eine endliche Abfolge von wohldefinierten Schritten und Berechnungen vorausgeht, sind diese Berechnungen Algorithmen. Sie werden mit Hilfe von Entwicklungsumgebungen in Computerprogramme implementiert, um sie auf dem jeweiligen Betriebssystem anwendbar zu gestalten. Aufgrund der Fülle an Detektoren für die *feature*-Erkennung bietet sich zur Unterscheidung eine Kategorisierung anhand der untersuchten Merkmale an [MIK08, S.216 ff.]. So ergeben sich die Suche nach Eckpunkten als *corner detectors*, die Suche nach Flecken als *blob detectors* und die Suche nach markanten Flächen als *region detectors*.

4.2 Corner Detectors

In diesem Kapitel, werden die am häufigsten in der Literatur erwähnten Algorithmen vorgestellt [MIK08, S.216 ff.]. Zu Beginn wird der *Harris corner detector*, mit seinem derivat-basierenden Ansatz, in 4.2.1 beschrieben. Es folgt der *SUSAN detector* unter 4.2.2, welcher ein Beispiel für die effizient morphologischen Operatoren darstellt. Abschließend erfolgt eine Diskussion über deren geeignete Anwendungen.

Corner detectors sind Programme, die aus einem Muster eines Bildes auf unterschiedliche Weise Ecken extrahieren können. Dabei ist der Begriff *corner* für „Ecke“ nicht wörtlich zu nehmen. Vielmehr entsprechen diesen gefundenen Punkten starke Krümmungen in einem Muster eines zweidimensionalen Bildes. Diese bedeuten keineswegs, dass dieses *feature* auch im dreidimensionalen Raum eine Ecke darstellt. Da ein Bild nicht im Gesamten, sondern nur als ein Bildausschnitt betrachtet wird, können auch in stark strukturierten Flächen *corners* extrahiert werden. Ob diese auch in der Realität Ecken darstellen, bleibt irrelevant. In der Literatur und Forschung gibt es eine ganze Reihe solcher Programme. Diese unterscheiden sich in ihrer zugrunde liegenden Extraktionstechnik und ihrer Ebene der *invariance*.

4.2.1 Harris detector

Der *Harris detector* wurde von Harris und Stephens entwickelt und ist oft auch als *Plessey* d*etector* bekannt. Er basiert auf der Idee, dass „ein Eckpunkt dort gegeben ist, wo der Gradient der Bildfunktion gleichzeitig in mehr als einer Richtung einen hohen Wert aufweist.“ [BUR05, S. 140 ff.]

Der *Harris detector* verwendet die ersten partiellen Ableitungen der Bildfunktion in horizontaler und vertikaler Richtung. Hierbei findet eine Nachbarschaftsuntersuchung eines Pixels in einem Bildausschnitt statt. Dabei sind nicht die absoluten Pixeldifferenzen entscheidend, sondern die Texturbeschaffenheit. Die Gradienten beschreiben den Farbverlauf des Untersuchungsraumes in Form von Farbwert-Differenzen und fassen diesen zu einem Vektor zusammen. Hat der Gradientenvektor nun eine markante Orientierung in eine Richtung, so wird dies als „Kante“ kategorisiert und nicht berücksichtigt. Treten mehrere markante Richtungen auf, so wird dieses Fens-

ter mit einer „komplexen Textur“ kategorisiert. Das heißt, dass speziell entlang von Kanten, wo der Gradient hoch, aber nur in einer Richtung ausgeprägt ist, dies nicht als Eckpunkt behandelt wird. Somit werden wirklich nur Ecken und Kreuzungen in Betracht gezogen. Es gibt zudem auch die Kategorie der „homogenen Textur“. Hier haben die Gradienten aller Pixel des Untersuchungsfensters geringe Werte. Dies tritt bei schwach-strukturierten Flächen auf. [BUR05, S.140 ff.]

Für die Suche nach *local features* ist nur die Kategorie der „komplexen Textur“ interessant. Nicht nur die Position, sondern auch der Gradientenvektor des Pixels beschreibt ein *local feature*. Somit kann dieses über mehrere Bilder identifizierbar sein. Da auch nur die Verhältnisse der Gradienten zueinander untersucht werden, sind solche Punkte translations- und rotationsinvariant. Zudem sind diese Punkte auch stabil gegenüber unterschiedlichen Lichtverhältnissen. Dies belegen mehrere Vergleichsstudien wie beispielsweise von SCHMID & MOHR [SCH10, S.19 ff.].

4.2.2 SUSAN detector

Der *SUSAN detector* wurde von SMITH & BRADY (1997) [SMI95, S.45 ff] eingeführt und baut auf verschiedene Techniken auf. *SUSAN* steht für *smallest univalue segment assimilating nucles* und „(…) vergleicht die Intensität der Pixel in einem kreisförmigen Fensterbereich mit dem Grauwert des zentralen Pixels, das als *Nucleus* bezeichnet wird“ [LUH10, S. 458]. Hierbei wird von jedem Pixel eines Grauwertbildes die Nachbarschaft kreisförmig untersucht. Dabei ist die Größe des Radius eine Konstante und wird während der Suche nicht verändert. Nun werden alle Pixel in diesem Radius in zwei Kategorien unterteilt, nämlich in „ähnlich“ oder „verschieden“, wobei bei der Zuordnung der Pixel die Intensität des *Nucleus* mit einem Schwellenwert genutzt wird.

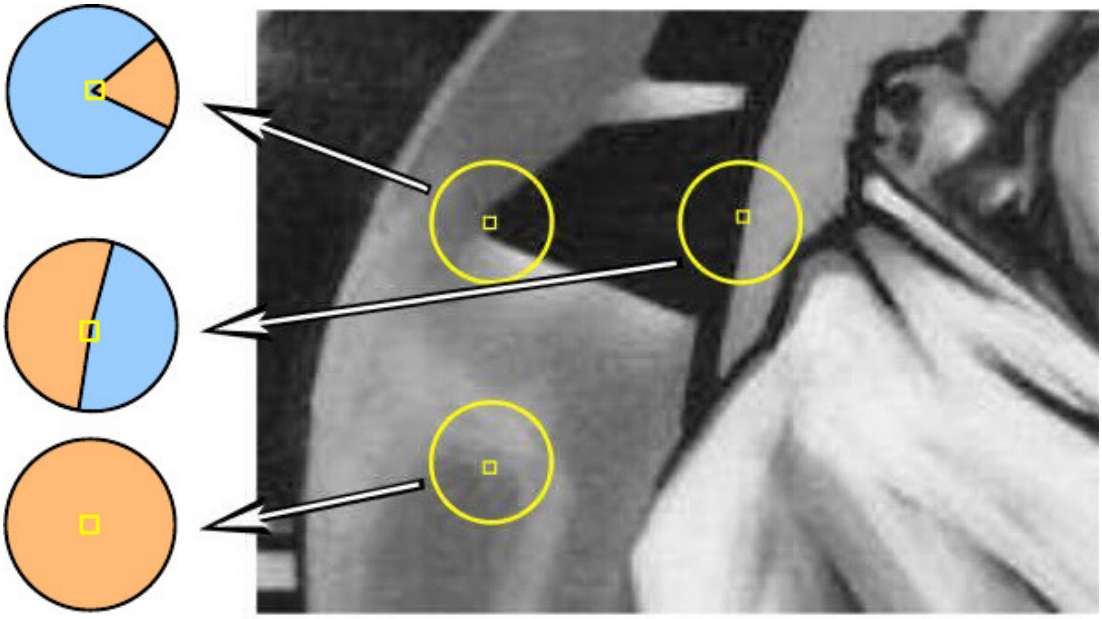


Abb. 10: SUSAN corners, (ähnlich = orange, verschieden = blau) (Quelle: [MIK08, S. 220])

Sind nun in diesem kreisförmigen Untersuchungsraum weniger als die Hälfte der Grauwerte einander ähnlich, so wird der *Nucleus* als Eckpunkt identifiziert. Dieser Punkt erhält als *feature*-Wert die Differenz aus der halben Anzahl der Pixel in dem Untersuchungsraum und die Summe der ähnlichen Pixel zum *Nucleus*. Auch dieser Detektor ist invariant gegenüber Translation und Rotation.

4.2.3 Diskussion *corner detectors*

Corner detectors identifizieren keine realen dreidimensionalen Eckpunkte, sondern nur eck-ähnliche Strukturen. Die beiden hier vorgestellten *corner detectors* sind in ihrer Grundfunktion ausschließlich invariant gegenüber Rotation und Translation, nicht aber gegenüber Maßstab und Affinität.

Viele Studien zeigen, dass der *Harris detector* die meisten stabilen *features* findet, aber der *SUSAN* robuster ist. Der *SUSAN* ist geeigneter, um *features* in harten Zeichnungen zu finden, aber nicht um ihn in realen Szenen einzusetzen. Um die *corner detectors* auch invariant gegenüber Maßstab und Affinität zu machen, müsste man ergänzende Algorithmen vorher anbringen.

4.3 Blob Detectors

Anders als bei den *corner detectors* zielen die *blob detectors* auf die Erkennung von Regionen in einem digitalen Bild ab. Ein *blob* ist eine Region eines digitalen Bildes, in dem einige Eigenschaften konstant sind oder innerhalb eines vorgegebenen Wertebereiches variieren. Eigenschaften können hierbei Helligkeit oder Farbe sein. Da es sich dabei um geschlossene Bereiche handelt, werden diese Detektoren in der Literatur unter anderem auch als *region extraction* bezeichnet. [MIK08, S.231 ff.]

Die BLOB-Erkennung ist eine übliche Methode des maschinellen Sehens und wird beispielsweise bei der Verfolgung von Gegenständen, für das Finden einer bestimmten Markierung oder bei der Erkennung von Personen eingesetzt.

4.3.1 Hessian Detector

Der *Hessian detector* oder Hessian-Matrix wurde im 19. Jahrhundert vom deutschen Mathematiker Ludwig Otto Hesse entwickelt und nach ihm benannt. Die Matrix beschreibt ausgehend von der Intensität des Bildes wie sich Formen in einem Bild gegenüber der Normalengleichung ändern. Hierbei werden die lokalen Maxima als BLOB-Struktur erkannt. [MIK08, S.231 ff.]

Die Hessian-Matrix wird unter anderem beim SURF-Algorithmus benutzt. [BAY08, S.1 ff.]

4.3.2 Intensity-based Regions

Der *intensity-based region detector* wurde von Tuytelaars und Van Gool [TUY04, S.61 ff.] zum Auffinden von affin-invarianten Regionen entwickelt. Dieser Detektor erkundet auf mehreren Ebenen und geht von Extremwerten der Bildintensitäten aus. Er untersucht daraufhin radial von diesem Punkt aus die Umgebung. Dies geschieht über eine Strahlenfunktion f(t). Die willkürliche Form wird durch ein Ellipsoid als *feature* ersetzt. [MIK08, S.238 ff.]

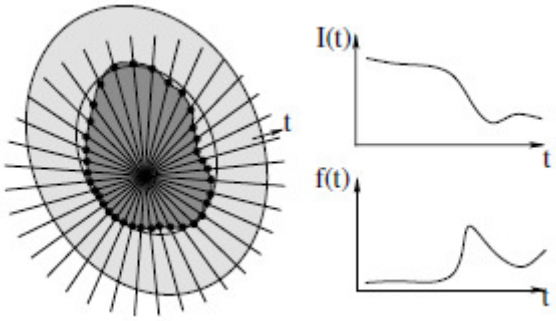


Abb. 11: Prinzip des intensity-based region detectors (I(t)= Intensität an Position t, f(t)= Funktion des Strahles) (Quelle: [MIK08, S. 239])

4.4 Kombinierte Detektoren

Kombinierte Detektoren sind Algorithmen, welche sowohl die Suche nach Merkmalen, als auch die Beschreibung dieser Merkmale durchführen können. In diesem Abschnitt werden die beiden bekanntesten und derzeit oft diskutierten Algorithmen SURF und SIFT näher beschrieben. Dabei wird diesmal auch kurz auf ihre Verfahren der Merkmalsbeschreibung eingegangen. Weitere kombinierte Algorithmen sind ORB (Oriented FAST and Rotated BRIEF vgl. [RUB11, S.1 ff.]) und FREAK (Fast Retina Keypoint vgl. [ALA12, S.1 ff.]).

4.4.1 SIFT

Der Scale Invariant Feature Transform (SIFT) ist ein Algorithmus, welcher unbeeindruckt von der Transformation und Skalierung der einzelnen Bilder Merkmale detektiert. Dabei geht er schrittweise vor. Im ersten Schritt werden in verschiedenen, mit dem Gauß-Algorithmus geglätteten Bildebenen, Extrema lokalisiert. [LOW04, S.2 ff.]

Anschließend werden ungeeignete Punkte, die z.B. zwischen zwei Pixel oder an Kanten liegen, aussortiert, da diese anfällig für Störungen sein können.

Im folgenden Schritt bekommen alle Merkmalspunkte eine Orientierung bei der auch der Glättungsgrad und die Bildebene als Information mit einfließen. So wird vermieden, dass ein einzelner Punkt mehrere Orientierungen bekommt. Es werden neue Punkte mit gleicher Position, aber unterschiedlicher Orientierung, gebildet. Dadurch können die Punkte der unterschiedlichen

Bildebenen extra behandelt und Korrespondenzen in schwach strukturierten Flächen gefunden werden. [LOW04, S.2 ff.]

Der SIFT bietet zusätzlich zu seiner Merkmalssuche auch eine lokale Beschreibung: den Merkmalspunktdeskriptor. Hier wird zu jedem erkannten Merkmalspunkt ein Deskriptor erstellt. Dadurch soll der Punkt eindeutig identifiziert werden können. Hierzu legt er einen 128-dimensionalen Vektor für jeden einzelnen Punkt an. Das macht ihn in seiner Arbeit sehr robust, aber auch sehr aufwändig in seiner Berechnung. [LOW04, S.14 ff.]

4.4.2 SURF

Speeded Up Robust Features (SURF) wurde von Bay et al. [BAY06, S.1 ff.] entwickelt und ist im groben eine Anpassung des SIFT. Er ist invariant gegenüber Maßstabsänderungen und Rotation. Er benutzt die Summe der Intensitätswerte die links bzw. oberhalb eines Pixels liegen.

Die Maßstabsunabhängigkeit wird nicht durch die Anwendung des Gauß-Filters in jeder Bildebene erreicht, sondern durch Änderung der SURF-Filter-Box bei Anwendung auf dem originalen Bild. Dadurch werden Merkmale schneller gefunden, da unter anderem die Berechnung der Bildpyramiden wegfällt. [BAY06, S.1 ff.]

Der SURF nutzt ebenfalls wie der SIFT eine Orientierung und Beschreibung der Merkmalspunkte. Diese besteht allerdings nur aus einem 64-dimensionalem Vektor. Die Kürze des Vektors sollte den SURF-Deskriptor im Vergleich zum SIFT schneller machen.

4.5 Bemerkungen

Von all den hier vorgestellten Verfahren sind SIFT und SURF wohl als die fortschrittlichsten anzusehen. Sie bieten eine hohe Stabilität gegenüber einer Vielzahl von Störungen, die bei der heutigen Technik der UAS immer auftreten können. Aufgrund der häufigen Verwendung von Kompaktkameras bei UAS, geschuldet der geringen Nutzlast, lassen sich viele Kameras nicht eindeutig kalibrieren. Darum eignen sich SIFT und SURF vor allem zur Zuordnung von Bildern, bei denen sich Maßstab, Rotation oder Perspektive stark

unterscheiden. Eine Kalibrierung der Kameraparameter wird somit nicht gebraucht. [LUH10, S. 160]

Merkmals Detektoren	Corner	Blob	Rotation invariant	Scale invariant	Affine invariant
Harris	√		√		
Hessian		√	√		
SUSAN	√		√		
Intensity-based Regions		√	√	√	√
SIFT	√	√	√	√	√
SURF	√	√	√	√	√

Tabelle 3: Übersicht über Invarianz der Detektoren (Quelle: [MIK08, S.257])

5. Methoden der Bildzuordnung mit OpenCV

Im Folgenden wird der Umgang mit OpenCV beschrieben und die fertigen Unterprogramme für eine erfolgreiche Bildzuordnung dieser Programmbibliothek dargeboten.

5.1 Warum OpenCV?

Die Abkürzung OpenCV steht für *Open Soucre Computer Vision Library*. *Computer Vision* kann frei übersetzt werden mit *maschinellem Sehen* und wird im Abschnitt 5.2 näher erläutert. Die OpenCV ist eine frei zugängliche und quelloffene Programmbibliothek mit einer Sammlung von Algorithmen und Unterprogrammen für das visuelle Sehen. Diese Unterprogramme können nicht eigenständig laufen. Die OpenCV Programmbibliothek ist in den Programmiersprachen C und C++ geschrieben und kann unter den Betriebssystemen Linux, Windows, Mac OS, iOS und Android verwendet werden [BRA08]. Die Entwicklung dieser Programmbibliothek wurde im Januar 1999 von Intel gestartet und wird heute hauptsächlich von Willow Garage, einem US-amerikanischen Unternehmen für Robotertechnologie aus Menlo Park, gepflegt. Derzeit werden fast 200.000 Downloads pro Monat registriert (Stand: 05.12.2013).

Die Stärken dieser Programmbibliothek liegen in der einfachen bereitgestellten Infrastruktur und in der Geschwindigkeit ihrer Funktionen. OpenCV ermöglicht es dadurch vielen Menschen, produktiver mit der Computer Vision arbeiten zu können. Hierbei liegt der Schwerpunkt ganz klar auf der *real-time vision*, also auf dem Echtzeitsehen. Dies erlaubt neben der Bewegungsverfolgung auch die Erkennung von Strukturen wie beispielsweise von Hindernissen oder Gestiken eines Gesichts und stellt eine große Unterstützung im Bereich des maschinellen Lernens dar. [BRA08, S.1 ff.]

Der Vorteil dieser Programmbibliothek im Vergleich zu anderen Computer Vision-Bibliotheken wie ImageJ[10] oder VXL ist, dass sie von Intel in C und C++ für Echtzeitanwendungen entwickelt wurde und somit ein Hardware-Hersteller eine Softwarebibliothek in einer auf Geschwindigkeit ausgelegten

[10] Link: http://rsb.info.nih.gov/ij/, von Wayne Rasband in Java entwickelt (National Institutes of Health, Bethesda (Maryland), USA)

Umgebung entwickelt hat. OpenCV ist Betriebssystem unabhängig und lässt sich daher auch auf Smartphones anwenden.

5.2 Was bedeutet Computer Vision?

Computer Vision ist ein Teilgebiet der Informatik und ein weitreichendes Feld der Bildinterpretation. In der Computer Vision vermischen sich die Fachgebiete der Informatik und der Mathematik. Auch gibt es hier Ähnlichkeiten zur Photogrammetrie, da sich einige mathematische Beziehungen gleichen, nur basieren diese auf unterschiedlichen Ausgangssituationen. Erst seit den letzten Jahren versucht man Problemstellungen aus der Photogrammetrie mit Hilfe der Computer Vision zu lösen.

Maschinelles Sehen trifft die Bedeutung dieses Ausdrucks nicht ganz. Vielmehr wird das optische Sehen von Kameras übernommen und allein der Dateninput wie Fotos oder Videos wird von Computerprogrammen interpretiert. Der Mensch gibt hierbei einen mathematischen Algorithmus vor, der nach konkreten Merkmalen ein Bild oder ein Video untersucht und legt im Vorfeld die gewünschten Kriterien fest. Hierbei wird versucht die Verarbeitung des menschlichen Gehirns von Gesehenem computergestützt nachzuempfinden [BRA08, S.1 ff.].

5.3 OpenCV-Bibliotheken

OpenCV bietet eine große Sammlung von mehr als 2500 Algorithmen[11] unter den Bedingungen der *Berkeley Software Distribution* (BSD), welche dadurch frei kopiert, verändert und verbreitet werden dürfen, wobei das ursprüngliche Copyright nicht entfernt werden darf. Diese Lizenzvereinbarung macht es möglich, diese Sammlung kommerziell zu nutzen. Hiervon ausgenommen sind zwei Algorithmen zur *feature point*-Extraktion und -Beschreibung. Die weiter oben schon beschriebenen SIFT- und SURF-Operatoren sind unter „*nonfree*“ in den Bibliotheken zu finden. Hier müssen die Lizenzbedingungen der Entwickler von SIFT und SURF beachtet werden. OpenCV liefert diese Algorithmen zum Evaluieren standardmäßig mit.

[11] Aktuelle Informationen sind unter: http://opencv.org (Stand: 13.12.2013) zu finden.

Die für den ersten Schritt der Verknüpfungspunktsuche relevanten Algorithmen sind in der Version 2.4.3 folgende *feature detectors*:

a. Harris
b. FAST
c. GFTT
d. SIFT
e. SURF
f. ORB
g. FREAK

Für die Punktbeschreibung als zweiten Schritt sind *folgende descriptor extractors* vorhanden:

a. BRIEF
b. SIFT
c. SURF
d. ORB
e. FREAK

Im dritten Schritt der Verknüpfungspunktsuche, dem Matching, finden sich folgende Algorithmen:

a. BruteForce
b. FlannBased
c. knnMatch

Leider können noch nicht alle diese Algorithmen miteinander kombiniert werden, da sie teilweise verschiedene Datentypen als Eingangs- oder Ausgangswert verlangen. Diese Einschränkung soll aber mit den kommenden OpenCV Versionen behoben werden.

6. Experimentelle Untersuchungen am Fallbeispiel

Um die Theorie der Korrespondenzanalyse unter OpenCV zu testen, werden einige experimentelle Untersuchungen an einem Fallbeispiel aus der Praxis vorgenommen. Dazu wird zuerst das Fallbeispiel vorgestellt und anschließend die Experimente mithilfe der Programmbibliotheken von OpenCV durchgeführt.

6.1 Fallbeispiel Auffahrten zur B188 bei Tangermünde

6.1.1 Projektbeschreibung

Die GEO-METRIK AG hat ihren Hauptsitz in der Nähe von Halle/Saale und ist ein weltweit agierender Dienstleister in den Hauptbereichen Vermessung und Geoinformatik. Im Bereich der Vermessung beschäftigt sich das Unternehmen unter anderem mit der Luftbildauswertung aus klassischen Befliegungen. Die GEO-METRIK ist bestrebt, neue Trends zeitnah aufzugreifen und diese für ihre Belange nutzbar zu machen. Aus diesem Grund ist die GEO-METRIK Ingenieurgesellschaft mbH Stendal seit März 2010 in dem Besitz eines Quadrocopters mit der Bezeichnung „md4-1000“.

Bei Projekten der GEO-METRIK hat es sich in der Vergangenheit bewährt, aktuelle Luftbilder als Zusatzinformationen für eine Bestandsvermessung oder als Planungsgrundlage dem Auftraggeber anzubieten. Durch diese zusätzlichen Informationen kann beispielsweise ein Planungsbüro bei der Sanierung eines Deiches Schadstellen sowie Wildtierüberläufe besser lokalisieren und darauf gezielter eingehen. Dadurch ergeben sich einerseits eine Zeitersparnis, da der Planer nicht mehr so oft vor Ort sein muss und andererseits eine Kostenersparnis, da somit Deichabschnitte frühzeitig gesondert befestigt werden können bevor es zu Schäden kommt und damit eine langfristige Stabilität gewährleistet wird.

Bei der Nutzung von Luftbildern als Planungsgrundlage ist es zudem von Vorteil, solche Bilder mit vorhandenen Geodaten zu verbinden und Übersichten zu erstellen, in denen der aktuelle unterirdische Stand der Leitungshaltung in Form von Lageplänen und der aktuelle oberirdische Stand mit Hilfe der Luftbilder hinterlegt werden. Um unterschiedliche Informationen mit Lage-

und Höhenbezug vergleichen oder verbinden zu können, müssen die Luftbilder vorerst georeferenziert werden.

Da bei ersten Projekten die klassischen Methoden der Auswertung nicht zum erhofften Erfolg geführt haben, wurde ein Testgebiet an der Bundesstraße 188 bei Tangermünde (Sachsen-Anhalt) unter anderem für diese Master Thesis beflogen.

Warum wurde genau dieses Testgebiet mit Straßenverlauf ausgewählt? Die Vermessung einer viel befahrenen Straße ist stellenweise nur durch Sperrungen der kompletten Fahrbahnen möglich. Dies kann durch den Einsatz von UAS in Zukunft vermieden werden, da diese über der Fahrbahn schweben und somit nicht den Straßenverkehr beeinflussen. In diesem Testgebiet sind zudem zahlreiche Oberflächenbeschaffenheiten und unterschiedliche Straßenobjekte vorhanden. So sind neben unterschiedlichen Straßenbelägen auch eine Brücke, Böschungen, Straßenschilder, Straßenmarkierungen und bewegte Objekte, die eventuell bei der Auswertung zu Problemen führen können, vorhanden.

Abb. 12: Untersuchungsgebiet Auffahrten B188 bei Tangermünde (Quelle Kartenmaterial: Google Earth)

6.1.2 Quadrocopter md4-1000

Bei dem hier verwendeten UAS handelt es sich um einen Quadrocopter der Firma Microdrones GmbH aus Siegen (Deutschland). Dieses Gerät nutzt vier akkubetriebene Rotoren als Antrieb. Jeder einzelne von ihnen wird mittels Bodenstation überwacht und kann per Fernbedienung händisch oder mittels Routenplanung gesteuert werden.

Abb. 13: Quadrocopter md4-1000 im Einsatz (Quelle: eigene Aufnahme)

Die md4-1000 ist mit einer Olympus Pen E-P2-Kamera und einem 17 mm Festbrennweiten-Objektiv ausgestattet. Diese Systemkamera wurde an der Beuth Hochschule für Technik in Berlin kalibriert. Hierbei wurde festgestellt, dass sich trotz deaktiviertem Autofokus und manueller Fokussierung auf Unendlich, die Kameraelektronik nicht gänzlich abgeschaltet werden kann und dadurch eine reproduzierbare Kalibrierung dieser Kamera nicht möglich ist [HER13, S. 104 ff.].

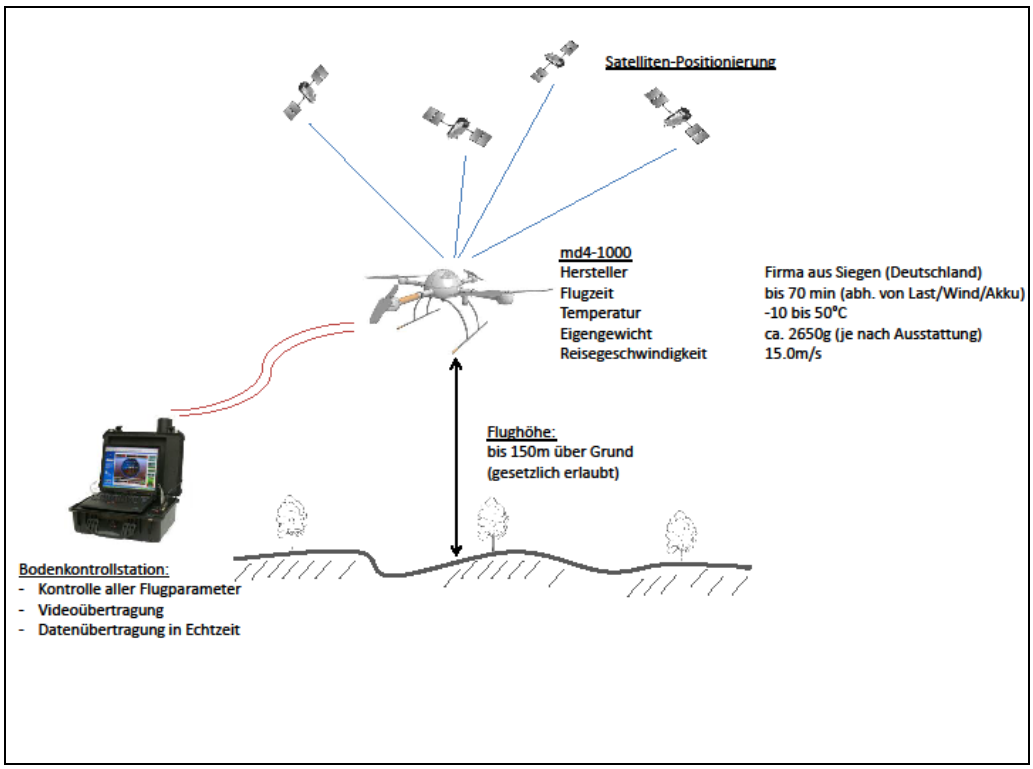


Abb. 14: Funktionsprinzip zur Steuerung des Quadrocopters (Quelle: eigener Entwurf)

Der Quadrocopter besitzt ein GPS- und ein IMU-Modul, welche das automatische Überfliegen von vorher definierten flächenförmigen oder einzelnen Aufnahmeobjekten ermöglichen. Mit Hilfe dieser Bauteile und einem Bordcomputer können vorher klar definierte Routen automatisch beflogen werden. Hierbei sind neben dem Starten und Landen auch die Kameraausrichtung und der Auslösezeitpunkt automatisiert möglich. Mittels dieser Routenplanung lassen sich größere Flächen wie auch in der klassischen Photogrammetrie rasterförmig befliegen. Diese rasterförmige Befliegung ist notwendig, um flächendeckende und in sich überlappende Aufnahmen zu garantieren.

Der Bordcomputer der Drohne nimmt zu jedem Zeitpunkt alle Sensorwerte auf und speichert diese auf einer externen Micro-SD-Karte. Pro Sekunde werden rund 120 Flugdaten aufgezeichnet. Damit lassen sich alle Flüge genauestens dokumentieren und gegebenenfalls nach Fehlern analysieren. In den Flugdaten werden unter anderem auch die Parameter der Lage der Drohne im 3D-Raum aufgezeichnet. Die Lageparameter sind *x, y, z, roll,*

pitch und *yaw*. Die Abb. 15 zeigt die Beschreibung der Lageparameter an einem Flugzeug, welche für ein UAS übertragbar sind.

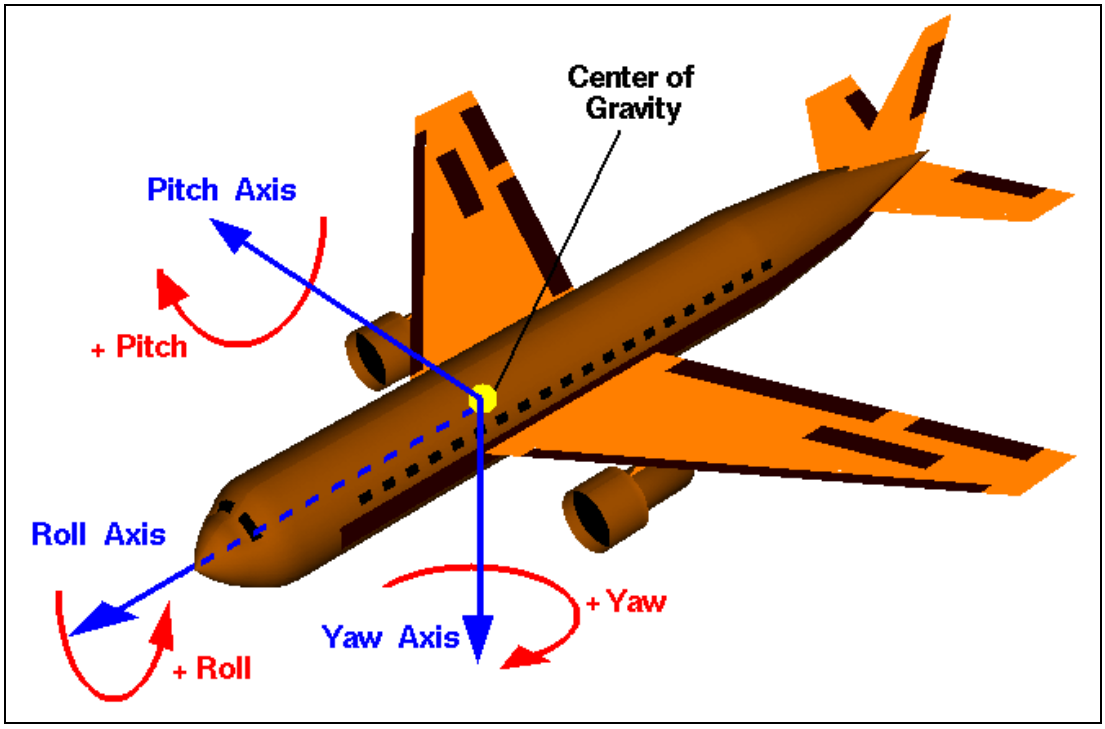


Abb. 15: Roll-Nick-Gier-Winkel, 3 Achsen zur Beschreibung der Lage eines Flugobjektes im 3D-Raum (Quelle: NASA, http://www.grc.nasa.gov/WWW/K-12/airplane/rotations.html, Stand: 04.05.2013)

6.1.3 Datenerhebung

In den nun folgenden Schritten wird komprimiert der Weg bis zu den Luftbildern beschrieben. An erster Stelle erfolgt die Flugplanung.

Eine möglichst genaue Flugplanung vor jedem Bildflug, sei es mit UAS oder Flugzeug, ist nicht nur wirtschaftlich enorm wichtig. Je genauer die Planung erfolgt, desto höhere Genauigkeiten können erzielt werden und tragen somit zur Wirtschaftlichkeit einer solchen Befliegung bei. Einem weiteren Flug zur Fehlerkorrektur kann somit vorgebeugt werden. Nach LUHMANN [LUH10] ist der erste Schritt der Flugplanung die Bestimmung der folgenden Projektparameter:

- Abmessung des Objektes
- zu erreichende Lagegenauigkeit

- Passpunktgenauigkeit
- Längs- und Querüberlappung
- Kamerakalibrierung
- Sensorparameter (Bildformat, Brennweite)
- Optische Parameter (Belichtungszeit, Tiefenschärfe)
- Bildmessgenauigkeit

Das Untersuchungsgebiet hat eine Größe von ca. 10 ha. Die Bodenauflösung beträgt 5 cm bei einer Flughöhe von 200 m. In Anlehnung an KRAUS [KRA04] und EISENBEIß [EIS09] wird eine Bildüberlappung in Längs- und Querrichtung von 80 % geplant. Der Grund, warum mit so hohen Überlappungen geplant wird, sind die Ungenauigkeiten der verwendeten GPS und IMU-Module des UAS. Die hohen Ungenauigkeiten der Rotationswinkel bei diesem Fallbeispiel, die teilweise über 40° betragen, hat HERDA nachgewiesen. [HER13, S.111 ff.]

Für diesen Flug wurden 9 Passpunke geplant. Aufgrund der Befahrung der Straße müssen Passpunkte immer der örtlichen Situation angepasst werden. Die Passpunkte werden mit GPS vermessen und mittels Korrekturparametern korrigiert. Somit ist die Lagegenauigkeit der Passpunkte niedriger als 3 cm.

Nach einer Besichtigung des Untersuchungsgebietes und Festlegung des Start- und Landepunktes, wird mit der Planung der Flugroute begonnen. Hierfür wird das mitgelieferte Programm „mdCockpit“ von Microdrones verwendet. Dieses Softwarepaket beinhaltet ein Unterpogramm zur Wegpunktplanung, den „Waypoint-Editor“, ein Überwachungsprogramm, den „Downlink-Decoder“, welches während des Fluges alle notwendigen Funktionen kontrolliert, und den Flugdatenschreiber, mit dem alle Flüge ausgewertet und gegebenenfalls Flugdaten exportiert werden können.

Als Planungsgrundlage nutzt der „Waypoint-Editor“ neben Bitmaps auch Google Earth, welches bei diesem Projekt genutzt wurde. Anschließend wird über den vordefinierten Flugbereich ein gleichmäßiges GIS-Raster gelegt.

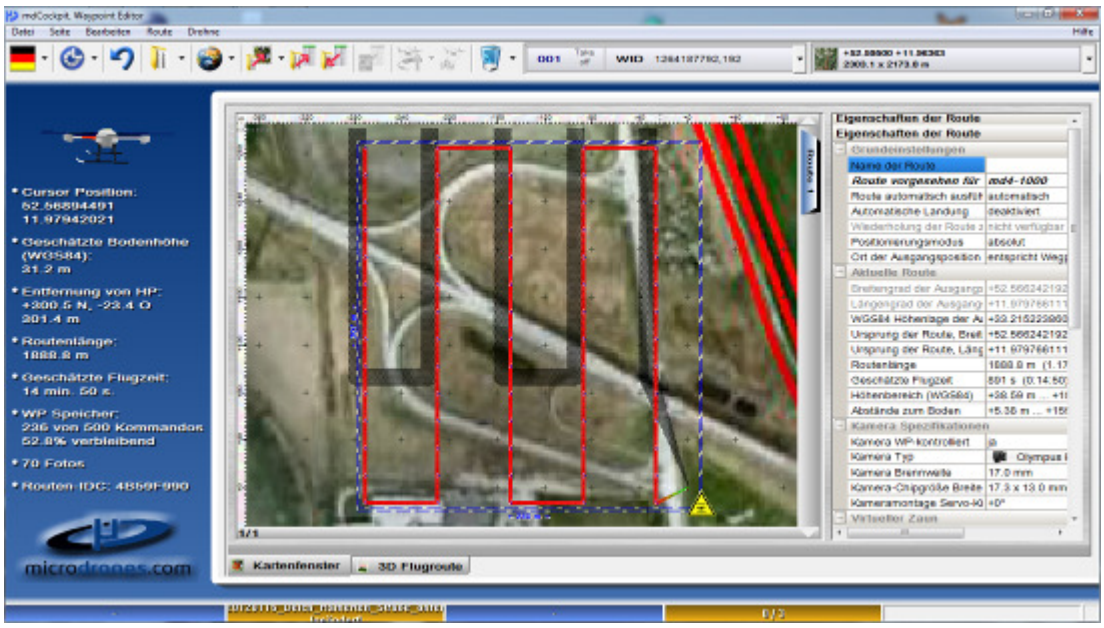


Abb. 16: Flugplanung mit GIS Raster B188 (Quelle: eigener Entwurf mit Hilfe von mdCockpit3.1)

Diese Flugplanung (Abb.16) kann nun auf die MicroSD-Karte des Quadrocopters exportiert werden.

Nach erfolgter Flugplanung wird die gewünschte Route beflogen.

6.1.4 Auswertung der Luftbilder

Ergebnisse der Befliegung vom 06.03.2012 sind zum einen 70 Fotos, welche auf der SD-Karte der Kamera gespeichert sind. Zum anderen entsteht eine Protokolldatei, in der unter anderem die Parameter der Fotostandpunkte gespeichert sind.

Mit den gespeicherten Flugdaten werden nun Orthomosaik und DOM erzeugt. Erste Versuche im März 2012 die die Auswertung mit LPS[12] vorsahen schlugen fehl, da LPS die Aerotriangulation nicht durchführen konnte. Grund hierfür waren falsch identifizierte Verknüpfungspunkte, bei der automatischen Suche. Eine manuelle Verknüpfungspunktsuche führte bei kleinen Bildverbänden zum Erfolg, ist allerdings sehr aufwendig.

[12] Leica Photogrammetry Suit: Professionelle Software zur klassischen Erzeugung von Oberflächenmodellen und Orthomosaike aus Luftbilder, http://geospatial.intergraph.com/products/imagine-photogrammetry/Details.aspx

6.2 Experimentelle Untersuchungen mit OpenCV

In diesem Abschnitt werden die Daten des Fallbeispiels experimentell mit OpenCV verarbeitet. Die Ergebnisse werden anschließend in LPS eingelesen und untersucht. Für die Verknüpfungspunktsuche kommen die Algorithmen zum Einsatz, die am wenigsten anfällig für Bildstörungen sind. In OpenCV ist der *Intensity-based Regions-Operator* nicht implementiert, daher werden nur SIFT und SURF verwendet.

6.2.1 Vorbereitungen

Am 11.11.2013 ist die neueste Version 2.4.7 (Stand: 12.12.2013) von OpenCV veröffentlicht worden. Die hier genutzte Version 2.4.3 wurde im November 2012 veröffentlicht. Die Funktionstüchtigkeit kann als stabil beschrieben werden. Auf der Internetseite[13] von OpenCV lassen sich sämtliche Versionen für den privaten sowie kommerziellen Gebrauch herunterladen. Für Windows liegt eine separate Installationsdatei vor.

Nach Abschluss der Installation ist das Setzen der System-Pfade für die Bibliotheken sinnvoll, wenn diese für eigene Programme genutzt werden sollen. Gegebenenfalls müssen die verwendeten DLL-Dateien[14] in jeden Projektordner kopiert werden.

Da OpenCV für C und C++ geschrieben ist, werden die experimentellen Programmcodes in C++ erstellt. Als Programmierwerkzeug wird die quelloffene Software *Eclipse IDE for C/C++ Developers*[15] verwendet.

6.2.2 Experimenteller Workflow

Die Idee besteht darin, einen Workflow unter der Verwendung unterschiedlicher Algorithmen zu erschaffen. Mit diesem soll die Verknüpfungspunktsuche in LPS ersetzt werden können. Der hier verwendete Workflow ist eine Anpassung eines vorhandenen Beispiels in OpenCV 2.4.3, welches bei der Installation mitgeliefert wird. Das Programmbeispiel heißt „mat-

[13] „http://opencv.org/downloads.html“
[14] DLL sind dynamische Programmbibliotheken zur Verwendung unter Windows.
[15] http://www.eclipse.org/downloads/moreinfo/c.php

ching_to_many_images.cpp“ und befindet sich nach der Installation von OpenCV im Unterordner „sample*“*.

Das Beispielprogramm von OpenCV benutzt ein Bild, welches es mit allen anderen Bildern paarweise vergleicht. Es werden in dem Abfragebild – auch *query image* genannt – *feature points* durch Verwendung von einem *extractor* und *descriptor* gesucht und beschrieben. Anschließend werden in den Folgebildern, den sogenannten *train images*, *feature points* extrahiert, beschrieben und mit den Merkmalsbeschreibungen des Abfragebildes mit einem *matching*-Algorithmus verglichen. Passen die Beschreibungen der Merkmale zusammen, werden die Bildkoordinaten der Punkte gespeichert. Zuletzt erfolgt eine Visualisierung der gefundenen Verknüpfungspunkte zur besseren Überprüfung. Hierbei werden die *feature points* mit einem gelben Kreis hervorgehoben und die gefundenen korrespondierenden Verknüpfungspunkte mit einer blauen Linie verbunden. Das Beispiel ist so aufgebaut, dass die benutzten Algorithmen durch Veränderung einer Variablen im Quelltext austauschbar sind. So lassen sich ohne große Modifikationen alle Untersuchungen schnellst möglich ausführen.

Die hier verwendeten Algorithmen zum Extrahieren und zum Beschreiben von *feature points* sind SIFT und SURF, da diese aktuell die am meisten analysierten und getesteten Detektoren und Deskriptoren sind. Zudem besitzen sie eine hohe Invarianz gegenüber Rotation, Rauschen, Maßstabs- und Helligkeitsänderungen. All diese Probleme können, eigenen praktischen Versuchen und Erfahrungen zufolge, bei Drohnenluftbildern auftauchen. Außerdem beschäftigen sich zahlreiche Internationale Universitäten wie die University of Otago in Neuseeland derzeit mit den Algorithmen und vergleichen diese miteinander. Hier steht allerdings die *Performance* während der Berechnung von *feature points* im Vordergrund. [KAH11, S. 501 ff]

Zum Filtern der korrekten Punkte wird bei beiden Experimenten der *FlannBasedMatcher* verwendet. Dieser überprüft die Merkmalsbeschreibungen der *feature points* und sucht zu einem Punkt den am nächsten gelegenen Punkt mit übereinstimmenden Merkmalen.

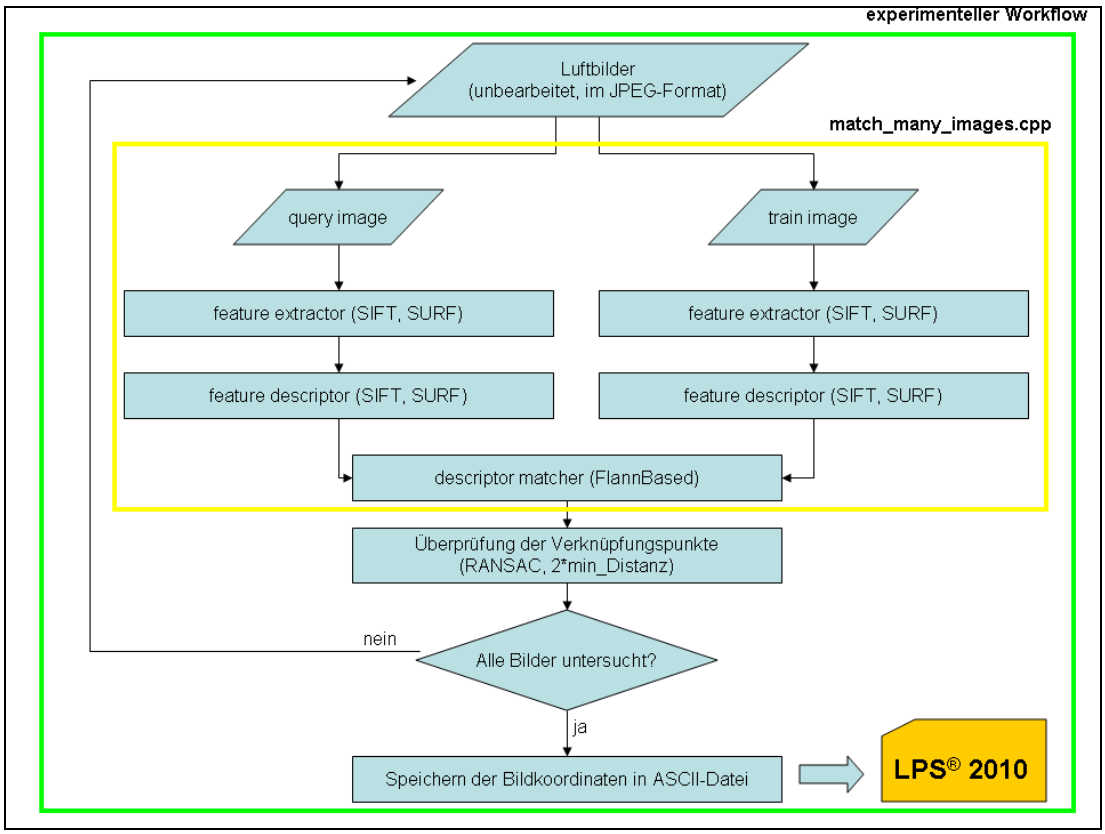


Abb. 17: experimenteller Workflow (Quelle: eigener Entwurf)

Anpassungen an das Programmbeispiel sind nun notwendig, um die erhaltenen Ergebnisse in LPS nutzen zu können. Als erstes werden die gefundenen Verknüpfungspunkte mit Hilfe von Zuordnungsverfahren gefiltert. In einigen anderen Beispielen von OpenCV sind zwei unterschiedliche Filtermethoden vorhanden. Zum einen ist das der unter 3.4.3 beschriebene RANSAC. Zum anderen kommt eine Filterung über die Distanz zum Einsatz. Bei der Distanzfilterung wird angenommen, dass der Abstand zwischen zwei verglichenen Merkmalen kleiner sein muss, als das Doppelte des kürzesten Abstandes von allen gefundenen Übereinstimmungen.

Damit LPS die Verknüpfungspunkinformationen nutzen kann, werden die Bildkoordinaten der Verknüpfungspunkte in eine Textdatei im ASCII-Format geschrieben. Dabei werden die Verknüpfungspunkte fortlaufend nummeriert und den Bildern, in denen sie vorkommen, zugewiesen. Nun werden noch die Bildkoordinaten der Verknüpfungspunkte, ausgehend von der linken oberen Ecke des Bildes, ausgegeben. Der vollständige Quelltext kann dem Anhang 1 entnommen werden.

Image ID	Point ID	X	Y
3063805	1	1727.320000	252.609000
3063805	2	3039.970000	2129.090000
3063805	3	3231.330000	2825.330000
3063805	4	906.070000	69.013900

Tabelle 4: Auszug aus einer Tie-Point-Datei aus LPS (Darstellung in Tabellenform) (Quelle: eigener Entwurf)

6.2.3 Daten

Für die Experimente werden nicht alle Fotos von dem Fallbeispiel verwendet, sondern nur ein Bildverband von 9 Bildern (3 x 3 Streifen) in voller Auflösung (4032 x 3024 Pixel). Die Überlappungen betragen 60 % in Längs- und 80 % in Querrichtung.

Abb. 18: Verwendeter Bildverband mit 60 % Längs- und 80 % Querüberlappung (Quelle: eigene Collage)

6.2.4 Vermutete Ergebnisse

Aufgrund der Invarianz beider Algorithmen wird vermutet, dass mit Hilfe der Bildkoordinaten eine Auswertung der Bildflüge im LPS durchzuführen ist. Darüber hinaus wird angenommen, dass SIFT und SURF mehr Verknüpfungspunkte in dem Bildverband finden als LPS. Eine hohe Anzahl an Verknüpfungspunkten bedeutet mehr Stabilität des Bildverbandes. Mit der Ver-

wendung der SIFT-Verknüpfungspunkte im LPS wird eine erfolgreiche Durchführung der Aerotriangulation im LPS erwartet.

6.2.5 Ergebnisse

Bei dem Programmdurchlauf unter Verwendung des SIFT Algorithmus kam es zu keinen Komplikationen. Alle verwendeten Bilder wurden erfolgreich ausgewertet. Es entstanden 66 paarweise Auswertungen mit 1609 korrekt ermittelten Verknüpfungspunkten. Bei der ersten Betrachtung der visuellen Ergebnisse der Auswertung wurde deutlich, dass die Filterung der *descriptor*-Zuordnungen über die Distanz, wie in Abb. 19 dargestellt, keine saubere Lösung ist.

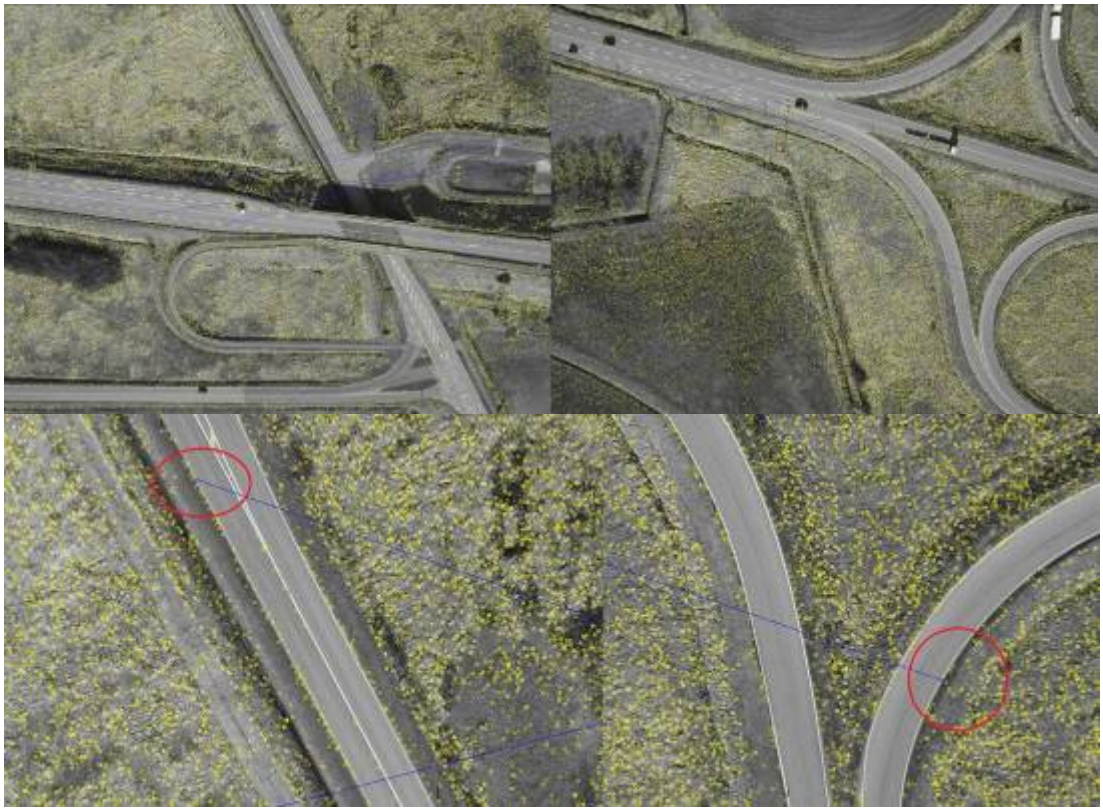


Abb. 19: falsche Zuordnung von SIFT-descriptors, oben: Paarweise Zuordnung, unten: falscher Verknüpfungspunkt im Detail (Quelle: eigener Entwurf)

Das ausgewertete Bildpaar mit insgesamt 11 übereinstimmenden Zuordnungen überlappt in Wirklichkeit nicht. Eigentlich dürften in diesen beiden Bildern keine Zuordnungen zueinander (blaue Linien) gefunden werden. Bei näherer Betrachtung der restlichen Ergebnisse wurde deutlich, dass ein großer Teil

der Bildzuordnungen nicht korrekt ist. Nur 18 der insgesamt 66 ausgewerteten Bildpaare konnten als „richtig“ eingestuft werden. In 37 Bildpaaren wurden keine übereinstimmenden Verknüpfungspunkte gefunden, obwohl eine ausreichende Überlappung gewährleistet ist. In 11 Bildpaaren entstanden „falsche“ Zuordnungen. Hierbei wurde deutlich, dass die meisten Fehlzuordnungen in der Auswertung der Bilder bei der Querüberlappung entstanden sind. Etwa 95% aller Zuordnungen in Querüberlappungsbereichen stimmen nicht. Hingegen stimmen alle Zuordnungen bei direkt benachbarten Bildern eines Flugstreifens in Längsrichtung. Alle paarweise ausgewerteten Bilder sind dieser Arbeit im Anhang 4 angehangen.

Die Bildkoordinaten der Verknüpfungspunkte wurden trotz der schlechten Beurteilung in LPS eingelesen. Anschließend erfolgte die Aerotriangulation, die jedoch keine verwertbaren Ergebnisse lieferte.

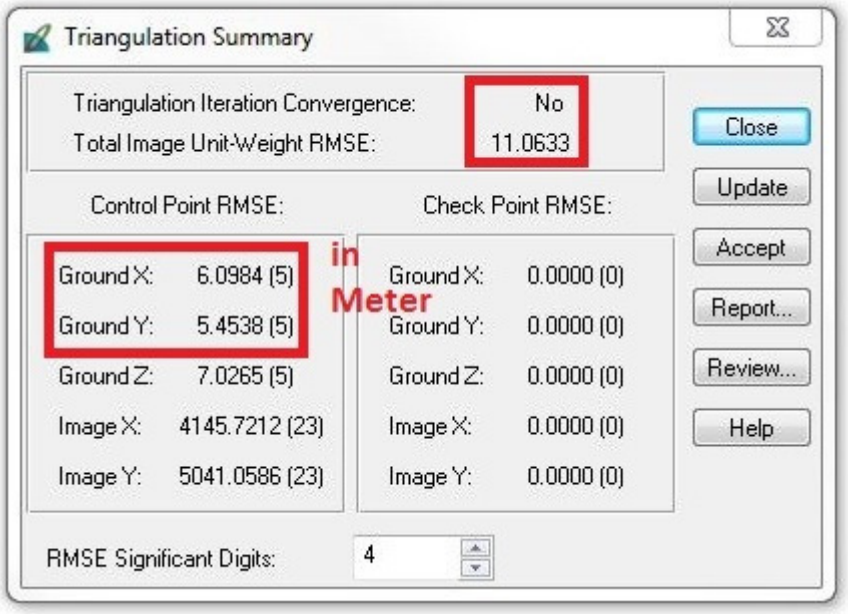


Abb. 20: Ergebnis der Aerotriangulation mit den SIFT-Verknüpungspunkten (Quelle: eigener Entwurf aus LPS 10)

Der Versuch, die Bibliothek des RANSAC einzubinden um die falschen Zuordnungen zu eliminieren, schlug fehl. Für die Einbindung liegt keine eindeutige Dokumentation vor. Während des Vorgangs der Kompilierung des Programmcodes, hing sich das Compiler-Programm MinGW[16] auf und der Computer musste neu gestartet werden. Auch eine komplette Neuinstallation der

[16] Ist ein Programm zum Erstellen ausführbarer Dateien unter Windows.

Software MinGW und OpenCV trug nicht zur Lösung des Problems bei. Somit wurde dieser Versuch abgebrochen. Der Quelltext befindet sich im Anhang 3.

Das gleiche Problem trat bei der Programmausführung mittels des SURF Algorithmus auf. Auch hier musste der Versuch abgebrochen werden und auch jegliche Recherche im Internet trug nicht zur Problembehebung bei. Dieser Quelltext befindet sich unter Anhang 2.

6.3 Beuteilung der Ergebnisse

Die OpenCV bietet mit ihren frei nutzbaren Programmbibliotheken ein großes Potenzial zum Erstellen von eigener Software für die Erstellung von Bildmosaiken oder Panoramabildern. Der große Vorteil ist die Echtzeitauswertung von Videosequenzen zur Objekterkennung und Objektverfolgung. Der Einsatz der Programmbibliotheken zur Geoinformationsgewinnung aus Luftbildern ist, wie experimentell belegt, unter alleiniger Verwendung der Bibliotheken nicht möglich. Die zuvor aufgestellten Vermutungen trafen nicht ein.

OpenCV ist ein mächtiges Werkzeug zum Erschaffen von Testumgebungen und zum Evaluieren einzelner Algorithmen. Nicht alle Programmbibliotheken sind für Gelegenheitsprogrammierer verständlich dokumentiert. Oft werden hier fundierte mathematische Kenntnisse der Bildbearbeitung und Programmierung vorausgesetzt.

Zwar sind sie online erhältlich, dennoch nur spärlich beschrieben. Hier lautet der Grundgedanke offensichtlich *„learning by doing“*. Allerdings gibt es auch Lehrbücher wie *Computer Vision with the OpenCV Library (2008)* von Bradski und Kaehler, in denen Schritt für Schritt Beispiele erklärt werden.

Diese Beurteilung soll nicht bedeuten, das OpenCV zur Programmierung von Auswertesoftware für Luftbilder nicht geeignet ist. Trotzdessen fehlt die Praxistauglichkeit, da einfach zu viel programmiertechnisches Verständnis erforderlich ist, um zu den erwarteten Ergebnissen zu gelangen. Dennoch ist OpenCV eine große Hilfe beim Umsetzen der erlernten Algorithmen und zudem geeignet, die Abläufe in der Computer Vision besser verstehen zu können.

Einen praxistauglichen Workflow kann es nach heutigem Stand ohne Weiteres noch nicht geben, da in der benutzten Version teilweise unterschiedliche Datentypen in den Algorithmen verwendet werden. Die Datentypen einheitlich zu programmieren ist sehr aufwändig und daher nicht praxistauglich.

Eine manuelle Verknüpfungspunktsuche im LPS ist zeitlich sehr aufwändig, führt aber in diesem Fallbeispiel zum Erfolg. In Abb. 21 ist die Auswertung der Aerotriangulation zu sehen.

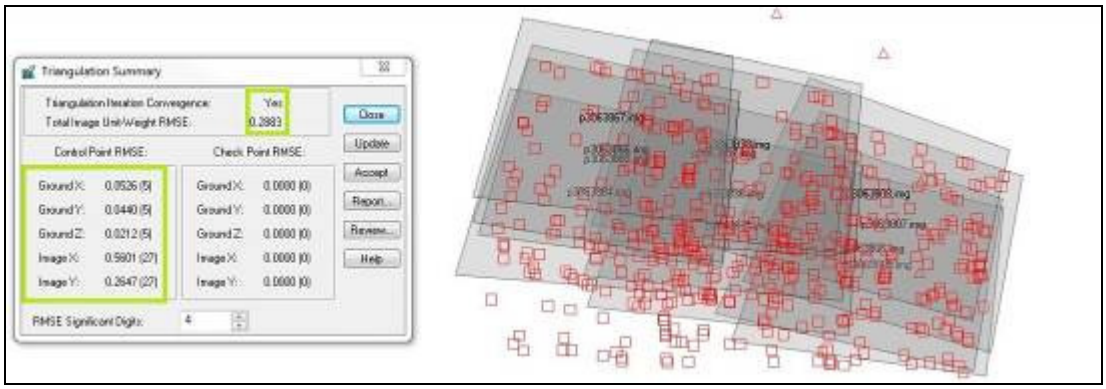


Abb. 21: Auswertung der Aerotriangulation in LPS (Quelle: eigener Entwurf aus LPS 10)

7. Resumé

7.1 Zusammenfassung

Der Grundgedanke zu Beginn dieser Arbeit war es, eine Möglichkeit zu finden, um die photogrammetrische Auswertung von Drohnenluftbildern unter Benutzung von LPS automatisiert zu ermöglichen. Zu diesem Zeitpunkt gab es noch keine ausgereifte kommerzielle Software zur Prozessierung von Drohnendaten. Mittlerweile ist der Einsatz von UAS in der täglichen Geoinformationsgewinnung ein gängiges Arbeitsmittel geworden beispielsweise bei der Erstellung von digitalen Oberflächenmodellen und Orthophotos zur Volumenberechnung von Abtragsflächen in Kieswerken. Nicht nur die Mirkotechnologie in den UAVs hat sich in den letzten Jahren stark weiter entwickelt, sondern auch die Softwareentwicklung. Dank der immer leistungsstärker werdenden Computer lassen sich heute Luftbilder fast vollautomatisch auswerten. Somit ist der Grundgedanke dieser Arbeit längst überholt, denn die Korrespondenzproblematik bei UAV-RGB-Daten kann im Allgemeinen als geklärt betrachtet werden. Es gibt leistungsstarke Softwares, welche auf UAV angepasste Algorithmen verwenden und die Auswertung der Luftbilddaten somit erleichtern und anwenderfreundlich gestalten.

Der SIFT und der SURF sind meiner Meinung nach die richtigen Ansätze, da die Sensortechnik derzeit noch nicht so genau ist, um korrekte Daten der Lage der UAVs zu liefern – jedenfalls nicht in den Gewichtsklassen wie es für den praktischen Gebrauch der Geoinformationsgewinnung sinnvoll wäre. Von daher muss immer mit Bildfehlern gerechnet werden. SIFT und SURF bieten normalerweise eine stabile Auswertemöglichkeit, die innerhalb dieser Versuchsreihe jedoch nicht erkennbar wurde.

7.2 Ausblicke

Generell kann festgestellt werden, dass es in der Theorie viele Lösungsansätze zur Lösung der Korrespondenz zwischen Drohnenluftbildern gibt. Praktische Umsetzungen findet man in einigen Kommerziellen Softwareprodukten wieder. Festzuhalten ist, dass sich die UAV-Branche enorm weiterentwickelt

und in naher Zukunft kein Stillstand zu erwarten ist, da eine hohe Nachfrage an diesem komplexem Thema besteht.

Selbst die Softwareanbieter der beiden bekanntesten Photogrammetrie-Softwareprodukten LPS und INPHO ziehen mit den UAV-Auswerteprodukten gleich und implementieren Programmelemente zur Auswertung für UAVs.

Zukünftig werden vermehrt d*ense matching* Algorithmen eingesetzt, die fast zu jedem Pixel einen 3D Punkt ermitteln können. Dadurch werden Objektoberflächen noch realen berechnet werden können. Wenn diese Verfahren eine Möglichkeit der Genauigkeitssteigerung gegenüber dem Laserscanner finden, wird der Einsatz von Kamerasystemen in der Vermessung um ein vielfaches zunehmen, da diese Technik auch auf Bilder vom Boden aus angewendet werden kann.

Denkbare Einsätze von OpenCV mit Drohnenbildern wären Programmentwicklungen, welche in Echtzeit eine Auswertung von aufgenommenen Fotos oder Videos rechnen, sodass der UAV-Pilot direkt nach seinem Flug eine grobe Auswertung der Aufnahmefläche erhält. Somit erhält er eine erste Aussage über sein Projekt. Auch ist eine vollautomatische 3D-Auswertung denkbar, indem ein Programm über Strukturerkennungsverfahren in den Fotos, selbsttätig Passpunktmarkenidentifiziert. Dies wird in der Nahbereichsphotogrammetrie schon heute praktiziert.

Abschließend ist festzuhalten, dass die Kombination von Drohnenluftbildern mit Algorithmen der *Computer Vision* unverzichtbar für die dreidimensionale Auswertung von Drohnendaten ist und diese für die Geodatengewinnung mit Hilfe bekannter photogrammetrischer Grundsätze sich weiterentwickeln wird und etablieren muss.

Literaturverzeichnis

[ALA12] Alahi, A., Ortiz, R. und Vandergheynst, P., 2012: *FREAK: Fast Retina Keypoint*. In: IEEE Conference on Computer Vision and Pattern Recognition,

aufgerufen am 06.06.2013 unter URL: http://infoscience.epfl.ch/record/175537/files/2069.pdf.

[BÄH91]Bähr, H.-P. und Vögtle, T., 1991: *Digitale Bildverarbeitung: Anwendung in Photogrammetrie, Kartographie und Fernerkundung*. 2. Auflage, Wichmann, Karlsruhe.

[BAY06] Bay, H., Tuytelaars, T und Van Gool, L., 2006: *SURF: Speeded up robust features*. In: *Proceedings of the European Conference on Computer Vision*, S. 404–417,

aufgerufen am 20.02.2013 unter URL: http://www.vision.ee.ethz.ch/~surf/eccv06.pdf.

[BUR05] Burger, W. und Burge, M.J., 2005: *Digitale Bildverarbeitung: Eine Einführung mit Java und ImageJ.* 2. Auflage, Springer, Heidelberg.

[BRA08] Bradski, G. und Kaehler, A., 2008: *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly, Sebastopol (USA).

[BRA07] Braun, H., 2007: *Aufstieg und Niedergang der Luftschifffahrt - Eine wirtschaftshistorische Analyse.* eurotrans-Verlag, Regensburg.

[EIS09] Eisenbeiß, H., 2009: *UAV Photogrammetry.* S.62-76, Zürich,

aufgerufen am 01.12.2012 unter URL: http://www.igp-data.ethz.ch/berichte/Blaue_Berichte_PDF/105.pdf.

[FIS81] Fischler, M. A. und Bolles, R. C., 1981: *Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography.* In: Communications of ACM. S. 381-395,

aufgerufen am 30.11.2012 unter URL: http://www.ai.sri.com/pubs/files/836.pdf

[HER13] Herda, H., Breuer, M., 2013: *Auswerteoptionen für Drohnenluftbilder – Versuch einer Bewertung*. In: 19. Workshop Computer-Bildanalyse in der Landwirtschaft, 2. Workshop Unbemannte autonom fliegende Systeme (UAS) in der Landwirtschaft. S.104-115.

[KAH11] Kahn, N. Y., McCane, B., Wyvill, G., 2011: *SIFT and SURF Performance Evaluation Against Various Image Deformations on Benchmar Dataset*, In: International Conference on Digital Image Computing: Techniques and Applications, S. 501-506.

aufgerufen am: 02.03.2013, unter URL: http://www.cs.otago.ac.nz/staffpriv/mccane/publications/nabeel2011Sift.pdf

[KRA04] Kraus, K., 2004: *Photogrammetrie, Band1, Geometrische Informationen aus Photographien und Laserscanneraufnahmen*, de Gruyter Lehrbuch, 7. Auflage, Berlin.

[LOW04] Lowe, D. G., 2004: *Distinctive Image Features from Scale-Invariant Keypoints*, In: International Journal of Computer Vision, S. 91-110, aufgerufen am 02.02.2013 unter URL: http://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf.

[LUH10] Luhmann, T., 2010: *Nahbereichsphotogrammetrie: Grundlagen, Methoden und Anwendungen*. Wichmann, Berlin.

[MIK08] Mikolajczyk, K. und Tuytelaars, T., 2008: *Local Invariant Feature Detectors: A Survey*,

aufgerufen am 30.01.2013 unter URL: http://epubs.surrey.ac.uk/726872/1/Tuytelaars-FGV-2008.pdf.

[RUB11] Rublee E., Rabaud, V., Konolige, K. und Bradski, G. R., 2011: *ORB: An efficient alternative to SIFT or SURF*. In: International Conference on Computer Vision, S. 2564-2571,

aufgerufen am 15.11.2013 unter URL: http://www.vision.cs.chubu.ac.jp/CV-R/pdf/Rublee_iccv2011.pdf.

[SCH10] Schmid, C., Mohr, R. und Bauckhage, C., 2010:. *Evaluation of Interest Point Detectors*. Montbonnot. In: International Journal of Coputer Vision, S. 151-172,

aufgerufen am 05.04.2013 unter URL: http://hal.inria.fr/docs/00/54/83/02/PDF/Schmid-ijcv00.pdf.

[SMI95] Smith, S. M. und Brady, J. M., 1995: *SUSAN - A New Approach to Low Levelimage Processing*. Oxford,

aufgerufen am 03.12.2013 unter URL: http://www.lems.brown.edu/vision/courses/image-processing/Readings/smith95susan.pdf.

[SNA06] Snavely, N., Seitz, S. M. und Szeliski, R., 2006: *Photo Tourism: Exploring image collections in 3D*. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2006),

aufgerufen am 02.03.2013 unter URL: http://phototour.cs.washington.edu/Photo_Tourism.pdf.

[STR11] Strecha, C., 2011: *Automated Photogrammetric Techniques on Ultra-light UAV Imagery*. S. 289-294,

aufgerufen am 20.04.2013 unter URL: http://www.ifp.uni-stuttgart.de/publications/phowo11/280Strecha.pdf.

[THA] Thamm, H.-P, o.A.: *UAV in der geographischen Feldforschung*,

aufgerufen am 20.12.2013 unter URL: http://dgk.auf.uni-rostock.de/uploads/media/17_Thamm_DFGRundgespraech.pdf.

[TUY04] Tuytelaars ,T. und Van Gool, L., 2004: *Matching widely separated views based on affine invariant regions*. In: International Journal of Computer Vision. S.61–85,

aufgerufen am 18.02.2013 unter URL: http://pdf.aminer.org/000/355/458/matching_of_affinely_invariant_regions_for_visual_servoing.pdf.

Anhang

Anhang 1: Quelltext des experimentellen Workflows

SIFT_SIFT_FlannBased.cpp

D:\UNIGIS\12 - MSTHESIS\01Daten\Experimente\results_SIFT_SIFT_FlannBased\SIFT_SIFT_FLANN.cpp Saturday, December 28, 2013 1:03 AM

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/features2d/features2d.hpp"
#include "opencv2/contrib/contrib.hpp"
#include "opencv2/nonfree/nonfree.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/calib3d/calib3d.hpp"
#include "opencv2/nonfree/features2d.hpp"

#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <fstream>

using namespace cv;
using namespace std;

const string defaultDetectorType = "SIFT";
const string defaultDescriptorType = "SIFT";
const string defaultMatcherType = "FlannBased";
const string defaultQueryImageName =
"D:/opencv_works/OpenCV243/test_matching_to_many_images_tangermuende/query_images/P3063865.JPG
";
const string defaultFileWithTrainImages =
"D:/opencv_works/OpenCV243/test_matching_to_many_images_tangermuende/train/trainImages.txt";
const string defaultDirToSaveResImages =
"D:/opencv_works/OpenCV243/test_matching_to_many_images_tangermuende/result";

int train_images_count = 0;
String queryfilename;

static void printPrompt( const string& applName )
{
    cout << "/*\n"
         << " * This is a sample on matching descriptors detected on one image to
descriptors detected in image set.\n"
         << " * So we have one query image and several train images. For each keypoint
descriptor of query image\n"
         << " * the one nearest train descriptor is found the entire collection of train
images. To visualize the result\n"
         << " * of matching we save images, each of which combines query and train image
with matches between them (if they exist).\n"
         << " * Match is drawn as line between corresponding points. Count of all matches is
equel to count of\n"
         << " * query keypoints, so we have the same count of lines in all set of result
images (but not for each result\n"
         << " * (train) image).\n"
         << " */\n" << endl;

    cout << endl << "Format:\n" << endl;
    cout << "./" << applName << " [detectorType] [descriptorType] [matcherType] [queryImage]
[fileWithTrainImages] [dirToSaveResImages]" << endl;
    cout << endl;

    cout << "\nExample:" << endl
         << "./" << applName << " " << defaultDetectorType << " " << defaultDescriptorType <<
 " " << defaultMatcherType << " "
         << defaultQueryImageName << " " << defaultFileWithTrainImages << " " <<
```

```
defaultDirToSaveResImages << endl;
}

static void maskMatchesByTrainImgIdx( const vector<DMatch>& matches, int trainImgIdx, vector<
char>& mask )
{
    mask.resize( matches.size() );
    fill( mask.begin(), mask.end(), 0 );
    for( size_t i = 0; i < matches.size(); i++ )
    {
        if( matches[i].imgIdx == trainImgIdx )
            mask[i] = 1;
    }
}

static void readTrainFilenames( const string& filename, string& dirName, vector<string>&
trainFilenames )
{
    trainFilenames.clear();

    ifstream file( filename.c_str() );
    if ( !file.is_open() )
        return;

    size_t pos = filename.rfind('\\');
    char dlmtr = '\\';
    if (pos == String::npos)
    {
        pos = filename.rfind('/');
        dlmtr = '/';
    }
    dirName = pos == string::npos ? "" : filename.substr(0, pos) + dlmtr;

    while( !file.eof() )
    {
        string str; getline( file, str );
        if( str.empty() ) break;
        trainFilenames.push_back(str);
        train_images_count = train_images_count + 1 ;
    }
    file.close();
}


static bool createDetectorDescriptorMatcher( const string& detectorType, const string&
descriptorType, const string& matcherType,
                                      Ptr<FeatureDetector>& featureDetector,
                                      Ptr<DescriptorExtractor>& descriptorExtractor,
                                      Ptr<DescriptorMatcher>& descriptorMatcher )
{
    cout << "< Creating feature detector, descriptor extractor and descriptor matcher ..." <<
 endl;
    featureDetector = FeatureDetector::create( detectorType );
    descriptorExtractor = DescriptorExtractor::create( descriptorType );
    descriptorMatcher = DescriptorMatcher::create( matcherType );
    cout << ">" << endl;
```

```
    bool isCreated = !( featureDetector.empty() || descriptorExtractor.empty() ||
descriptorMatcher.empty() );
    if( !isCreated )
        cout << "Can not create feature detector or descriptor extractor or descriptor
matcher of given types." << endl << ">" << endl;

    return isCreated;
}

static bool readImages( const string& queryImageName, const string& trainFilename,
                Mat& queryImage, vector <Mat>& trainImages, vector<string>& trainImageNames )
{
    cout << "< Reading the images..." << endl;
    queryImage = imread( queryImageName, CV_LOAD_IMAGE_GRAYSCALE);
    if( queryImage.empty() )
    {
        cout << "Query image can not be read. No image found!" << endl << ">" << endl;
        return false;
    }
    string trainDirName;
    readTrainFilenames( trainFilename, trainDirName, trainImageNames );
    if( trainImageNames.empty() )
    {
        cout << "Train image filenames can not be read." << endl << ">" << endl;
        return false;
    }
    int readImageCount = 0;
    for( size_t i = 0; i < trainImageNames.size(); i++ )
    {
        string filename = trainDirName + trainImageNames[i];
        Mat img = imread( filename, CV_LOAD_IMAGE_GRAYSCALE );
        if( img.empty() )
            cout << "Train image " << filename << " can not be read." << endl;
        else
            readImageCount++;
        trainImages.push_back( img );
    }
    if( !readImageCount )
    {
        cout << "All train images can not be read." << endl << ">" << endl;
        return false;
    }
    else
        cout << readImageCount << " train images were read." << endl;
    cout << ">" << endl;

    return true;
}

static void detectKeypoints( const Mat& queryImage, vector<KeyPoint>& queryKeypoints,
                     const vector<Mat>& trainImages, vector<vector<KeyPoint> >&
trainKeypoints,
                     Ptr<FeatureDetector>& featureDetector )
{
    cout << endl << "< Extracting keypoints from images..." << endl;
    featureDetector->detect( queryImage, queryKeypoints );
    featureDetector->detect( trainImages, trainKeypoints );
```

```
    cout << ">" << endl;
}

static void computeDescriptors( const Mat& queryImage, vector<KeyPoint>& queryKeypoints, Mat&
 queryDescriptors,
                                const vector<Mat>& trainImages, vector<vector<KeyPoint> >&
trainKeypoints, vector<Mat>& trainDescriptors,
                                Ptr<DescriptorExtractor>& descriptorExtractor )
{
    cout << "< Computing descriptors for keypoints..." << endl;
    descriptorExtractor->compute( queryImage, queryKeypoints, queryDescriptors );
    descriptorExtractor->compute( trainImages, trainKeypoints, trainDescriptors );

    int totalTrainDesc = 0;
    for( vector<Mat>::const_iterator tdIter = trainDescriptors.begin(); tdIter !=
trainDescriptors.end(); tdIter++ )
        totalTrainDesc += tdIter->rows;

    cout << "Query descriptors count: " << queryDescriptors.rows << "; Total train
descriptors count: " << totalTrainDesc << endl;
    cout << ">" << endl;
}

static void matchDescriptors( const Mat& queryDescriptors, const vector<Mat>&
trainDescriptors,
                              vector<DMatch>& matches, Ptr<DescriptorMatcher>& descriptorMatcher )
{
    cout << "< Set train descriptors collection in the matcher and match query descriptors
to them..." << endl;
    TickMeter tm;

    tm.start();
    descriptorMatcher->add( trainDescriptors );
    descriptorMatcher->train();
    tm.stop();
    double buildTime = tm.getTimeMilli();

    tm.start();
    descriptorMatcher->match( queryDescriptors, matches );
    tm.stop();
    double matchTime = tm.getTimeMilli();

    CV_Assert( queryDescriptors.rows == (int)matches.size() || matches.empty() );

    cout << "Number of matches: " << matches.size() << endl;
    cout << "Build time: " << buildTime << " ms; Match time: " << matchTime << " ms" << endl;
    cout << ">" << endl;
}

static void saveResultImages( const Mat& queryImage, const vector<KeyPoint>& queryKeypoints,
                              const vector<Mat>& trainImages, const vector<vector<KeyPoint> >&
trainKeypoints,
                              const vector<DMatch>& matches, const vector<string>& trainImagesNames,
 const string& resultDir )
{
    cout << "< Save results..." << endl;
    Mat drawImg;
```

```
    vector<char> mask;
    for( size_t i = 0; i < trainImages.size(); i++ )
    {
        if( !trainImages[i].empty() )
        {
            maskMatchesByTrainImgIdx( matches, (int)i, mask );
            drawMatches( queryImage, queryKeypoints, trainImages[i], trainKeypoints[i],
                         matches, drawImg, Scalar(255, 0, 0), Scalar(0, 255, 255), mask );
            string filename = resultDir + "/" + queryfilename + "_" + trainImagesNames[i];
            if( !imwrite( filename, drawImg ) )
                cout << "Image " << filename << " can not be saved (may be because directory
" << resultDir << " does not exist)." << endl;
        }
    }
    cout << ">" << endl;
}

int main(int argc, char** argv)
{
    string detectorType = defaultDetectorType;
    string descriptorType = defaultDescriptorType;
    string matcherType = defaultMatcherType;
    string queryImageName = defaultQueryImageName;
    string fileWithTrainImages = defaultFileWithTrainImages;
    string dirToSaveResImages = defaultDirToSaveResImages;


    if( argc != 7 && argc != 1 )
    {
        printPrompt( argv[0] );
        return -1;
    }

    if( argc != 1 )
    {
        detectorType = argv[1]; descriptorType = argv[2]; matcherType = argv[3];
        queryImageName = argv[4]; fileWithTrainImages = argv[5];
        dirToSaveResImages = argv[6];
    }

    initModule_nonfree();

    Ptr<FeatureDetector> featureDetector;
    Ptr<DescriptorExtractor> descriptorExtractor;
    Ptr<DescriptorMatcher> descriptorMatcher;
    if( !createDetectorDescriptorMatcher( detectorType, descriptorType, matcherType,
featureDetector, descriptorExtractor, descriptorMatcher ) )
    {
        printPrompt( argv[0] );
        return -1;
    }

    Mat queryImage;
    vector<Mat> trainImages;
    vector<string> trainImagesNames;
    if( !readImages( queryImageName, fileWithTrainImages, queryImage, trainImages,
trainImagesNames ) )
```

```
    {
        printPrompt( argv[0] );
        return -1;
    }

    vector<KeyPoint> queryKeypoints;
    vector<vector<KeyPoint> > trainKeypoints;
    detectKeypoints( queryImage, queryKeypoints, trainImages, trainKeypoints, featureDetector
);

    Mat queryDescriptors;
    vector<Mat> trainDescriptors;
    computeDescriptors( queryImage, queryKeypoints, queryDescriptors,
                        trainImages, trainKeypoints, trainDescriptors,
                        descriptorExtractor );

    vector<DMatch> matches;
    matchDescriptors( queryDescriptors, trainDescriptors, matches, descriptorMatcher );


   /* saveResultImages( queryImage, queryKeypoints, trainImages, trainKeypoints,
                      matches, trainImagesNames, dirToSaveResImages );*/


    //Begin TW-Test
        double max_dist = 0; double min_dist = 100;
        //-- Quick calculation of max and min distances between keypoints
         for( int i = 0; i < queryDescriptors.rows; i++ )
         { double dist = matches[i].distance;
           if( dist < min_dist ) min_dist = dist;
           if( dist > max_dist ) max_dist = dist;
         }

        //-- Draw only "good" matches (i.e. whose distance is less than 2*min_dist )
          //-- PS.- radiusMatch can also be used here.
          std::vector< DMatch > good_matches;

          for( int i = 0; i < queryDescriptors.rows; i++ )
          { if( matches[i].distance < 2*min_dist )
            { good_matches.push_back( matches[i]); }
          }
          for( int i = 0; i < (int)good_matches.size(); i++ )
           {
              int q = good_matches[i].queryIdx;
              int t = good_matches[i].trainIdx;
              int ti = train_images_count;
              cout << "-- Good Match [" << i << "] on " << matches[i].imgIdx << "_x: " <<
                      queryKeypoints[q].pt.x << ti << "_y: " << queryKeypoints[q].pt.y << "
-- "<< trainImagesNames[0] << endl;
                      count << " -- " << trainImages[1] << "_x: "<< trainKeypoints[0][t].pt.x
<< " " << trainImages[1] << "_y: " << trainKeypoints[0][0].pt.y << endl;
              //trainKeypoints[2][t].pt.x --> geht
           }

            //End TW-Test
```

```
        saveResultImages( queryImage, queryKeypoints, trainImages, trainKeypoints,
                          good_matches, trainImagesNames, dirToSaveResImages );
    return 0;

}
```

Anhang 2: Quelltext des experimentellen Workflows

SURF_SURF_FlannBased.cpp

D:\UNIGIS\12 - MSTHESIS\01Daten\Experimente\results_SURF_SURF_FlannBased\SURF_SURF_FlannBased.cpp Monday, December 30, 2013 6:26 PM

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/features2d/features2d.hpp"
#include "opencv2/contrib/contrib.hpp"
#include "opencv2/nonfree/nonfree.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/calib3d/calib3d.hpp"
#include "opencv2/nonfree/features2d.hpp"

#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <fstream>

using namespace cv;
using namespace std;

const string defaultDetectorType = "SURF";
const string defaultDescriptorType = "SURF";
const string defaultMatcherType = "FlannBased";
const string defaultQueryImageName =
"D:/opencv_works/OpenCV243/test_matching_to_many_images_tangermuende/query_images/P3063865.JPG
";
const string defaultFileWithTrainImages =
"D:/opencv_works/OpenCV243/test_matching_to_many_images_tangermuende/train/trainImages.txt";
const string defaultDirToSaveResImages =
"D:/opencv_works/OpenCV243/test_matching_to_many_images_tangermuende/result";

int train_images_count = 0;
String queryfilename;

static void printPrompt( const string& applName )
{
    cout << "/*\n"
         << " * This is a sample on matching descriptors detected on one image to
descriptors detected in image set.\n"
         << " * So we have one query image and several train images. For each keypoint
descriptor of query image\n"
         << " * the one nearest train descriptor is found the entire collection of train
images. To visualize the result\n"
         << " * of matching we save images, each of which combines query and train image
with matches between them (if they exist).\n"
         << " * Match is drawn as line between corresponding points. Count of all matches is
equel to count of\n"
         << " * query keypoints, so we have the same count of lines in all set of result
images (but not for each result\n"
         << " * (train) image).\n"
         << " */\n" << endl;

    cout << endl << "Format:\n" << endl;
    cout << "./" << applName << " [detectorType] [descriptorType] [matcherType] [queryImage]
[fileWithTrainImages] [dirToSaveResImages]" << endl;
    cout << endl;

    cout << "\nExample:" << endl
         << "./" << applName << " " << defaultDetectorType << " " << defaultDescriptorType <<
 " " << defaultMatcherType << " "
         << defaultQueryImageName << " " << defaultFileWithTrainImages << " " <<
```

```
defaultDirToSaveResImages << endl;
}

static void maskMatchesByTrainImgIdx( const vector<DMatch>& matches, int trainImgIdx, vector<
char>& mask )
{
    mask.resize( matches.size() );
    fill( mask.begin(), mask.end(), 0 );
    for( size_t i = 0; i < matches.size(); i++ )
    {
        if( matches[i].imgIdx == trainImgIdx )
            mask[i] = 1;
    }
}

static void readTrainFilenames( const string& filename, string& dirName, vector<string>&
trainFilenames )
{
    trainFilenames.clear();

    ifstream file( filename.c_str() );
    if ( !file.is_open() )
        return;

    size_t pos = filename.rfind('\\');
    char dlmtr = '\\';
    if (pos == String::npos)
    {
        pos = filename.rfind('/');
        dlmtr = '/';
    }
    dirName = pos == string::npos ? "" : filename.substr(0, pos) + dlmtr;

    while( !file.eof() )
    {
        string str; getline( file, str );
        if( str.empty() ) break;
        trainFilenames.push_back(str);
        train_images_count = train_images_count + 1 ;
    }
    file.close();
}


static bool createDetectorDescriptorMatcher( const string& detectorType, const string&
descriptorType, const string& matcherType,
                                      Ptr<FeatureDetector>& featureDetector,
                                      Ptr<DescriptorExtractor>& descriptorExtractor,
                                      Ptr<DescriptorMatcher>& descriptorMatcher )
{
    cout << "< Creating feature detector, descriptor extractor and descriptor matcher ..." <<
 endl;
    featureDetector = FeatureDetector::create( detectorType );
    descriptorExtractor = DescriptorExtractor::create( descriptorType );
    descriptorMatcher = DescriptorMatcher::create( matcherType );
    cout << ">" << endl;
```

```
    bool isCreated = !( featureDetector.empty() || descriptorExtractor.empty() ||
descriptorMatcher.empty() );
    if( !isCreated )
        cout << "Can not create feature detector or descriptor extractor or descriptor
matcher of given types." << endl << ">" << endl;

    return isCreated;
}

static bool readImages( const string& queryImageName, const string& trainFilename,
                Mat& queryImage, vector <Mat>& trainImages, vector<string>& trainImageNames )
{
    cout << "< Reading the images..." << endl;
    queryImage = imread( queryImageName, CV_LOAD_IMAGE_GRAYSCALE);
    if( queryImage.empty() )
    {
        cout << "Query image can not be read. No image found!" << endl << ">" << endl;
        return false;
    }
    string trainDirName;
    readTrainFilenames( trainFilename, trainDirName, trainImageNames );
    if( trainImageNames.empty() )
    {
        cout << "Train image filenames can not be read." << endl << ">" << endl;
        return false;
    }
    int readImageCount = 0;
    for( size_t i = 0; i < trainImageNames.size(); i++ )
    {
        string filename = trainDirName + trainImageNames[i];
        Mat img = imread( filename, CV_LOAD_IMAGE_GRAYSCALE );
        if( img.empty() )
            cout << "Train image " << filename << " can not be read." << endl;
        else
            readImageCount++;
        trainImages.push_back( img );
    }
    if( !readImageCount )
    {
        cout << "All train images can not be read." << endl << ">" << endl;
        return false;
    }
    else
        cout << readImageCount << " train images were read." << endl;
    cout << ">" << endl;

    return true;
}

static void detectKeypoints( const Mat& queryImage, vector<KeyPoint>& queryKeypoints,
                     const vector<Mat>& trainImages, vector<vector<KeyPoint> >&
trainKeypoints,
                     Ptr<FeatureDetector>& featureDetector )
{
    cout << endl << "< Extracting keypoints from images..." << endl;
    featureDetector->detect( queryImage, queryKeypoints );
    featureDetector->detect( trainImages, trainKeypoints );
```

```
    cout << ">" << endl;
}

static void computeDescriptors( const Mat& queryImage, vector<KeyPoint>& queryKeypoints, Mat&
 queryDescriptors,
                           const vector<Mat>& trainImages, vector<vector<KeyPoint> >&
trainKeypoints, vector<Mat>& trainDescriptors,
                           Ptr<DescriptorExtractor>& descriptorExtractor )
{
    cout << "< Computing descriptors for keypoints..." << endl;
    descriptorExtractor->compute( queryImage, queryKeypoints, queryDescriptors );
    descriptorExtractor->compute( trainImages, trainKeypoints, trainDescriptors );

    int totalTrainDesc = 0;
    for( vector<Mat>::const_iterator tdIter = trainDescriptors.begin(); tdIter !=
trainDescriptors.end(); tdIter++ )
        totalTrainDesc += tdIter->rows;

    cout << "Query descriptors count: " << queryDescriptors.rows << "; Total train
descriptors count: " << totalTrainDesc << endl;
    cout << ">" << endl;
}

static void matchDescriptors( const Mat& queryDescriptors, const vector<Mat>&
trainDescriptors,
                      vector<DMatch>& matches, Ptr<DescriptorMatcher>& descriptorMatcher )
{
    cout << "< Set train descriptors collection in the matcher and match query descriptors
to them..." << endl;
    TickMeter tm;

    tm.start();
    descriptorMatcher->add( trainDescriptors );
    descriptorMatcher->train();
    tm.stop();
    double buildTime = tm.getTimeMilli();

    tm.start();
    descriptorMatcher->match( queryDescriptors, matches );
    tm.stop();
    double matchTime = tm.getTimeMilli();

    CV_Assert( queryDescriptors.rows == (int)matches.size() || matches.empty() );

    cout << "Number of matches: " << matches.size() << endl;
    cout << "Build time: " << buildTime << " ms; Match time: " << matchTime << " ms" << endl;
    cout << ">" << endl;
}

static void saveResultImages( const Mat& queryImage, const vector<KeyPoint>& queryKeypoints,
                      const vector<Mat>& trainImages, const vector<vector<KeyPoint> >&
trainKeypoints,
                      const vector<DMatch>& matches, const vector<string>& trainImagesNames,
 const string& resultDir )
{
    cout << "< Save results..." << endl;
    Mat drawImg;
```

```
    vector<char> mask;
    for( size_t i = 0; i < trainImages.size(); i++ )
    {
        if( !trainImages[i].empty() )
        {
            maskMatchesByTrainImgIdx( matches, (int)i, mask );
            drawMatches( queryImage, queryKeypoints, trainImages[i], trainKeypoints[i],
                         matches, drawImg, Scalar(255, 0, 0), Scalar(0, 255, 255), mask );
            string filename = resultDir + "/" + queryfilename + "_" + trainImagesNames[i];
            if( !imwrite( filename, drawImg ) )
                cout << "Image " << filename << " can not be saved (may be because directory
" << resultDir << " does not exist)." << endl;
        }
    }
    cout << ">" << endl;
}

int main(int argc, char** argv)
{
    string detectorType = defaultDetectorType;
    string descriptorType = defaultDescriptorType;
    string matcherType = defaultMatcherType;
    string queryImageName = defaultQueryImageName;
    string fileWithTrainImages = defaultFileWithTrainImages;
    string dirToSaveResImages = defaultDirToSaveResImages;


    if( argc != 7 && argc != 1 )
    {
        printPrompt( argv[0] );
        return -1;
    }

    if( argc != 1 )
    {
        detectorType = argv[1]; descriptorType = argv[2]; matcherType = argv[3];
        queryImageName = argv[4]; fileWithTrainImages = argv[5];
        dirToSaveResImages = argv[6];
    }

    initModule_nonfree();

    Ptr<FeatureDetector> featureDetector;
    Ptr<DescriptorExtractor> descriptorExtractor;
    Ptr<DescriptorMatcher> descriptorMatcher;
    if( !createDetectorDescriptorMatcher( detectorType, descriptorType, matcherType,
featureDetector, descriptorExtractor, descriptorMatcher ) )
    {
        printPrompt( argv[0] );
        return -1;
    }

    Mat queryImage;
    vector<Mat> trainImages;
    vector<string> trainImagesNames;
    if( !readImages( queryImageName, fileWithTrainImages, queryImage, trainImages,
trainImagesNames ) )
```

```
    {
        printPrompt( argv[0] );
        return -1;
    }

    vector<KeyPoint> queryKeypoints;
    vector<vector<KeyPoint> > trainKeypoints;
    detectKeypoints( queryImage, queryKeypoints, trainImages, trainKeypoints, featureDetector
);

    Mat queryDescriptors;
    vector<Mat> trainDescriptors;
    computeDescriptors( queryImage, queryKeypoints, queryDescriptors,
                        trainImages, trainKeypoints, trainDescriptors,
                        descriptorExtractor );

    vector<DMatch> matches;
    matchDescriptors( queryDescriptors, trainDescriptors, matches, descriptorMatcher );


   /* saveResultImages( queryImage, queryKeypoints, trainImages, trainKeypoints,
                      matches, trainImagesNames, dirToSaveResImages );*/


    //Begin TW-Test
        double max_dist = 0; double min_dist = 100;
        //-- Quick calculation of max and min distances between keypoints
         for( int i = 0; i < queryDescriptors.rows; i++ )
         { double dist = matches[i].distance;
           if( dist < min_dist ) min_dist = dist;
           if( dist > max_dist ) max_dist = dist;
         }

        //-- Draw only "good" matches (i.e. whose distance is less than 2*min_dist )
          //-- PS.- radiusMatch can also be used here.
          std::vector< DMatch > good_matches;

          for( int i = 0; i < queryDescriptors.rows; i++ )
          { if( matches[i].distance < 2*min_dist )
            { good_matches.push_back( matches[i]); }
          }
          for( int i = 0; i < (int)good_matches.size(); i++ )
           {
              int q = good_matches[i].queryIdx;
              int t = good_matches[i].trainIdx;
              int ti = train_images_count;
              cout << "-- Good Match [" << i << "] on " << matches[i].imgIdx << "_x: " <<
                     queryKeypoints[q].pt.x << ti << "_y: " << queryKeypoints[q].pt.y << "
-- "<< trainImagesNames[0] << endl;
                     //count << " -- " << trainImages[1] << "_x: "<<
trainKeypoints[0][t].pt.x << " " << trainImages[1] << "_y: " << trainKeypoints[0][0].pt.y <<
endl;
              //trainKeypoints[2][t].pt.x --> geht
           }

            //End TW-Test
```

```
        saveResultImages( queryImage, queryKeypoints, trainImages, trainKeypoints,
                          good_matches, trainImagesNames, dirToSaveResImages );
    return 0;

}
```

Anhang 3: Quelltext des experimentellen Workflows

SIFT_SIFT_FlannBased_RANSAC.cpp

D:\UNIGIS\12 - MSTHESIS\01Daten\Experimente\results_SURF_SURF_FlannBased\SURF_SURF_FlannBased_RANSAC.cpp Monday, December 30, 2013 6:36 PM

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/features2d/features2d.hpp"
#include "opencv2/contrib/contrib.hpp"
#include "opencv2/nonfree/nonfree.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/calib3d/calib3d.hpp"
#include "opencv2/nonfree/features2d.hpp"

#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <fstream>

using namespace cv;
using namespace std;

const string defaultDetectorType = "SURF";
const string defaultDescriptorType = "SURF";
const string defaultMatcherType = "FlannBased";
const string defaultQueryImageName =
"D:/opencv_works/OpenCV243/test_matching_to_many_images_tangermuende/query_images/P3063865.JPG
";
const string defaultFileWithTrainImages =
"D:/opencv_works/OpenCV243/test_matching_to_many_images_tangermuende/train/trainImages.txt";
const string defaultDirToSaveResImages =
"D:/opencv_works/OpenCV243/test_matching_to_many_images_tangermuende/result";

int train_images_count = 0;
String queryfilename;

static void printPrompt( const string& applName )
{
    cout << "/*\n"
         << " * This is a sample on matching descriptors detected on one image to
descriptors detected in image set.\n"
         << " * So we have one query image and several train images. For each keypoint
descriptor of query image\n"
         << " * the one nearest train descriptor is found the entire collection of train
images. To visualize the result\n"
         << " * of matching we save images, each of which combines query and train image
with matches between them (if they exist).\n"
         << " * Match is drawn as line between corresponding points. Count of all matches is
equel to count of\n"
         << " * query keypoints, so we have the same count of lines in all set of result
images (but not for each result\n"
         << " * (train) image).\n"
         << " */\n" << endl;

    cout << endl << "Format:\n" << endl;
    cout << "./" << applName << " [detectorType] [descriptorType] [matcherType] [queryImage]
[fileWithTrainImages] [dirToSaveResImages]" << endl;
    cout << endl;

    cout << "\nExample:" << endl
         << "./" << applName << " " << defaultDetectorType << " " << defaultDescriptorType <<
 " " << defaultMatcherType << " "
         << defaultQueryImageName << " " << defaultFileWithTrainImages << " " << "
```

```
defaultDirToSaveResImages << endl;
}

static void maskMatchesByTrainImgIdx( const vector<DMatch>& matches, int trainImgIdx, vector<
char>& mask )
{
    mask.resize( matches.size() );
    fill( mask.begin(), mask.end(), 0 );
    for( size_t i = 0; i < matches.size(); i++ )
    {
        if( matches[i].imgIdx == trainImgIdx )
            mask[i] = 1;
    }
}

static void readTrainFilenames( const string& filename, string& dirName, vector<string>&
trainFilenames )
{
    trainFilenames.clear();

    ifstream file( filename.c_str() );
    if ( !file.is_open() )
        return;

    size_t pos = filename.rfind('\\');
    char dlmtr = '\\';
    if (pos == String::npos)
    {
        pos = filename.rfind('/');
        dlmtr = '/';
    }
    dirName = pos == string::npos ? "" : filename.substr(0, pos) + dlmtr;

    while( !file.eof() )
    {
        string str; getline( file, str );
        if( str.empty() ) break;
        trainFilenames.push_back(str);
        train_images_count = train_images_count + 1 ;
    }
    file.close();
}


static bool createDetectorDescriptorMatcher( const string& detectorType, const string&
descriptorType, const string& matcherType,
                                      Ptr<FeatureDetector>& featureDetector,
                                      Ptr<DescriptorExtractor>& descriptorExtractor,
                                      Ptr<DescriptorMatcher>& descriptorMatcher )
{
    cout << "< Creating feature detector, descriptor extractor and descriptor matcher ..." <<
 endl;
    featureDetector = FeatureDetector::create( detectorType );
    descriptorExtractor = DescriptorExtractor::create( descriptorType );
    descriptorMatcher = DescriptorMatcher::create( matcherType );
    cout << ">" << endl;
```

```
    bool isCreated = !( featureDetector.empty() || descriptorExtractor.empty() ||
descriptorMatcher.empty() );
    if( !isCreated )
        cout << "Can not create feature detector or descriptor extractor or descriptor
matcher of given types." << endl << ">" << endl;

    return isCreated;
}

static bool readImages( const string& queryImageName, const string& trainFilename,
                Mat& queryImage, vector <Mat>& trainImages, vector<string>& trainImageNames )
{
    cout << "< Reading the images..." << endl;
    queryImage = imread( queryImageName, CV_LOAD_IMAGE_GRAYSCALE);
    if( queryImage.empty() )
    {
        cout << "Query image can not be read. No image found!" << endl << ">" << endl;
        return false;
    }
    string trainDirName;
    readTrainFilenames( trainFilename, trainDirName, trainImageNames );
    if( trainImageNames.empty() )
    {
        cout << "Train image filenames can not be read." << endl << ">" << endl;
        return false;
    }
    int readImageCount = 0;
    for( size_t i = 0; i < trainImageNames.size(); i++ )
    {
        string filename = trainDirName + trainImageNames[i];
        Mat img = imread( filename, CV_LOAD_IMAGE_GRAYSCALE );
        if( img.empty() )
            cout << "Train image " << filename << " can not be read." << endl;
        else
            readImageCount++;
        trainImages.push_back( img );
    }
    if( !readImageCount )
    {
        cout << "All train images can not be read." << endl << ">" << endl;
        return false;
    }
    else
        cout << readImageCount << " train images were read." << endl;
    cout << ">" << endl;

    return true;
}

static void detectKeypoints( const Mat& queryImage, vector<KeyPoint>& queryKeypoints,
                     const vector<Mat>& trainImages, vector<vector<KeyPoint> >&
trainKeypoints,
                     Ptr<FeatureDetector>& featureDetector )
{
    cout << endl << "< Extracting keypoints from images..." << endl;
    featureDetector->detect( queryImage, queryKeypoints );
    featureDetector->detect( trainImages, trainKeypoints );
```

```
    cout << ">" << endl;
}

static void computeDescriptors( const Mat& queryImage, vector<KeyPoint>& queryKeypoints, Mat&
 queryDescriptors,
                           const vector<Mat>& trainImages, vector<vector<KeyPoint> >&
trainKeypoints, vector<Mat>& trainDescriptors,
                           Ptr<DescriptorExtractor>& descriptorExtractor )
{
    cout << "< Computing descriptors for keypoints..." << endl;
    descriptorExtractor->compute( queryImage, queryKeypoints, queryDescriptors );
    descriptorExtractor->compute( trainImages, trainKeypoints, trainDescriptors );

    int totalTrainDesc = 0;
    for( vector<Mat>::const_iterator tdIter = trainDescriptors.begin(); tdIter !=
trainDescriptors.end(); tdIter++ )
        totalTrainDesc += tdIter->rows;

    cout << "Query descriptors count: " << queryDescriptors.rows << "; Total train
descriptors count: " << totalTrainDesc << endl;
    cout << ">" << endl;
}

static void matchDescriptors( const Mat& queryDescriptors, const vector<Mat>&
trainDescriptors,
                      vector<DMatch>& matches, Ptr<DescriptorMatcher>& descriptorMatcher )
{
    cout << "< Set train descriptors collection in the matcher and match query descriptors
to them..." << endl;
    TickMeter tm;

    tm.start();
    descriptorMatcher->add( trainDescriptors );
    descriptorMatcher->train();
    tm.stop();
    double buildTime = tm.getTimeMilli();

    tm.start();
    descriptorMatcher->match( queryDescriptors, matches );
    tm.stop();
    double matchTime = tm.getTimeMilli();

    CV_Assert( queryDescriptors.rows == (int)matches.size() || matches.empty() );

    cout << "Number of matches: " << matches.size() << endl;
    cout << "Build time: " << buildTime << " ms; Match time: " << matchTime << " ms" << endl;
    cout << ">" << endl;
}

static void saveResultImages( const Mat& queryImage, const vector<KeyPoint>& queryKeypoints,
                      const vector<Mat>& trainImages, const vector<vector<KeyPoint> >&
trainKeypoints,
                      const vector<DMatch>& matches, const vector<string>& trainImagesNames,
 const string& resultDir )
{
    cout << "< Save results..." << endl;
    Mat drawImg;
```

```
    vector<char> mask;
    for( size_t i = 0; i < trainImages.size(); i++ )
    {
        if( !trainImages[i].empty() )
        {
            maskMatchesByTrainImgIdx( matches, (int)i, mask );
            drawMatches( queryImage, queryKeypoints, trainImages[i], trainKeypoints[i],
                         matches, drawImg, Scalar(255, 0, 0), Scalar(0, 255, 255), mask );
            string filename = resultDir + "/" + queryfilename + "_" + trainImagesNames[i];
            if( !imwrite( filename, drawImg ) )
                cout << "Image " << filename << " can not be saved (may be because directory
" << resultDir << " does not exist)." << endl;
        }
    }
    cout << ">" << endl;
}

int main(int argc, char** argv)
{
    string detectorType = defaultDetectorType;
    string descriptorType = defaultDescriptorType;
    string matcherType = defaultMatcherType;
    string queryImageName = defaultQueryImageName;
    string fileWithTrainImages = defaultFileWithTrainImages;
    string dirToSaveResImages = defaultDirToSaveResImages;


    if( argc != 7 && argc != 1 )
    {
        printPrompt( argv[0] );
        return -1;
    }

    if( argc != 1 )
    {
        detectorType = argv[1]; descriptorType = argv[2]; matcherType = argv[3];
        queryImageName = argv[4]; fileWithTrainImages = argv[5];
        dirToSaveResImages = argv[6];
    }

    initModule_nonfree();

    Ptr<FeatureDetector> featureDetector;
    Ptr<DescriptorExtractor> descriptorExtractor;
    Ptr<DescriptorMatcher> descriptorMatcher;
    if( !createDetectorDescriptorMatcher( detectorType, descriptorType, matcherType,
featureDetector, descriptorExtractor, descriptorMatcher ) )
    {
        printPrompt( argv[0] );
        return -1;
    }

    Mat queryImage;
    vector<Mat> trainImages;
    vector<string> trainImagesNames;
    if( !readImages( queryImageName, fileWithTrainImages, queryImage, trainImages,
trainImagesNames ) )
```

```
    {
        printPrompt( argv[0] );
        return -1;
    }

    vector<KeyPoint> queryKeypoints;
    vector<vector<KeyPoint> > trainKeypoints;
    detectKeypoints( queryImage, queryKeypoints, trainImages, trainKeypoints, featureDetector
);

    Mat queryDescriptors;
    vector<Mat> trainDescriptors;
    computeDescriptors( queryImage, queryKeypoints, queryDescriptors,
                        trainImages, trainKeypoints, trainDescriptors,
                        descriptorExtractor );

    vector<DMatch> matches;
    matchDescriptors( queryDescriptors, trainDescriptors, matches, descriptorMatcher );


   /* saveResultImages( queryImage, queryKeypoints, trainImages, trainKeypoints,
                      matches, trainImagesNames, dirToSaveResImages );*/


    //Begin TW-Test
        double max_dist = 0; double min_dist = 100;
        //-- Quick calculation of max and min distances between keypoints
         for( int i = 0; i < queryDescriptors.rows; i++ )
         { double dist = matches[i].distance;
           if( dist < min_dist ) min_dist = dist;
           if( dist > max_dist ) max_dist = dist;
         }

        //-- Draw only "good" matches (i.e. whose distance is less than 2*min_dist )
          //-- PS.- radiusMatch can also be used here.
          std::vector< DMatch > good_matches;

          for( int i = 0; i < queryDescriptors.rows; i++ )
          { if( matches[i].distance < 2*min_dist )
            { good_matches.push_back( matches[i]); }
          }
          /*for( int i = 0; i < (int)good_matches.size(); i++ )
           {
              int q = good_matches[i].queryIdx;
              int t = good_matches[i].trainIdx;
              int ti = train_images_count;
              cout << "-- Good Match [" << i << "] on " << matches[i].imgIdx << "_x: " <<
                     queryKeypoints[q].pt.x << ti << "_y: " << queryKeypoints[q].pt.y << "
-- "<< trainImagesNames[0] << endl;
                     //count << " -- " << trainImages[1] << "_x: "<<
trainKeypoints[0][t].pt.x << " " << trainImages[1] << "_y: " << trainKeypoints[0][0].pt.y <<
endl;
              //trainKeypoints[2][t].pt.x --> geht
           }*/
          //-- Localize the object from img_1 in img_2
          std::vector<Point2f> obj;
          std::vector<Point2f> scene;
```

```
          for( int i = 0; i < good_matches.size(); i++ )
          {
            //-- Get the keypoints from the good matches
            obj.push_back( keypoints_object[ good_matches[i].queryIdx ].pt );
            scene.push_back( keypoints_scene[ good_matches[i].trainIdx ].pt );
          }

          Mat H = findHomography( obj, scene, CV_RANSAC );

          //-- Get the corners from the image_1 ( the object to be "detected" )
          std::vector<Point2f> obj_corners(4);
          obj_corners[0] = cvPoint(0,0); obj_corners[1] = cvPoint( img_object.cols, 0 );
          obj_corners[2] = cvPoint( img_object.cols, img_object.rows ); obj_corners[3] =
cvPoint( 0, img_object.rows );
          std::vector<Point2f> scene_corners(4);

          perspectiveTransform( obj_corners, scene_corners, H);


          //-- Draw lines between the corners (the mapped object in the scene - image_2 )
          line( img_matches, scene_corners[0] + Point2f( img_object.cols, 0), scene_corners[1
] + Point2f( img_object.cols, 0), Scalar(0, 255, 0), 4 );
          line( img_matches, scene_corners[1] + Point2f( img_object.cols, 0), scene_corners[2
] + Point2f( img_object.cols, 0), Scalar( 0, 255, 0), 4 );
          line( img_matches, scene_corners[2] + Point2f( img_object.cols, 0), scene_corners[3
] + Point2f( img_object.cols, 0), Scalar( 0, 255, 0), 4 );
          line( img_matches, scene_corners[3] + Point2f( img_object.cols, 0), scene_corners[0
] + Point2f( img_object.cols, 0), Scalar( 0, 255, 0), 4 );

            //End TW-Test



          saveResultImages( queryImage, queryKeypoints, trainImages, trainKeypoints,
                            good_matches, trainImagesNames, dirToSaveResImages );
    return 0;

}
```

Anhang 4: Ergebnisse: paarweise Punktzuordnung nach SIFT_SIFT_FlannBased