Diplomarbeit

Objektorientierte Bildverarbeitungsalgorithmen zum relativen Hovern eines autonomen Helikopters

von Lukas Goormann

Studiengang Medieninformatik
Fachhochschule Braunschweig/Wolfenbüttel
Matrikelnummer 20018218

fertiggestellt am 17. Dezember 2004

Zusammenfassung

In Rahmen dieser Diplomarbeit werden Strategien und Bildverabeitungsalgorithmen zur Realisierung des „relativen Hoverns“ eines autonomen Kleinhelikopters untersucht. Unter relativem Hovern versteht man dabei das Positionhalten, relativ zu Objekten und Untergründen, die von einer Kamera aufgenommen werden. Insbesondere wurden Blob-Analyse-Algorithmen zu dieser Problemstellung entwickelt und geprüft, ob und in wie weit die Blob-Analyse zur Bestimmung der Eigenbewegung geeignet ist.

Des Weiteren wurden ein objektorientiertes Framework, Konzepte und Applikationen zum Realisieren und Testen von Bildverarbeitungsalgorithmen auf verschiedenen Plattformen erstellt. Mit Hilfe dieses Frameworks wurden auch einige der Blob-Analyse-Algorithmen implementiert.

Erste Tests haben ergeben, dass sich die Blob-Analyse zum „relativen Hovern“ unter „Echtzeit“-Bedingungen gut eignet, da sie sehr schnell und dabei ähnlich genau wie andere zur Zeit verwendete Algorithmen ist.

Inhaltsverzeichnis

Vorwort und eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich diese Arbeit vollständig selbst verfasst, keine anderen als die genannten Quellen und keine unerlaubten Hilfsmittel benutzt habe.

Dennoch basiert diese Diplomarbeit natürlich auch auf meinen Erfahrungen und dem Stoff zahlreicher Vorlesungen meines Studiums der Medieninformatik an der Fachhochschule-Braunschweig/Wolfenbüttel.

Vor allem Gedanken und Konzepte aus den Vorlesungen zur Softwaretechnik von Professor Dr. rer. nat. Pekrun, Grafische Datenverarbeitung und Data Warehouse/Data Mining bei Professor Dr. rer. nat. habil. Klawonn und Digitale Bildverarbeitung bei Dr.-Ing. Schneider und der zugehörigen Laborveranstaltung unter Mitarbeit von Dipl.-Ing. Stolpmann haben viele der hier verwendeten Ideen und Algorithmen geprägt und inspiriert. Dies gilt insbesondere für letztgenannte Veranstaltung, in deren Rahmen ich erstmalig das Konzept des Blobs als Datenstruktur für „Objekte“ in einem Bild und die Verwendung der Blob-Analyse zur Mustererkennung (allerdings in statischen, zeitinvarianten Bildern) kennen gelernt habe, ohne welche diese Arbeit eine gänzlich andere Form angenommen hätte. All diesen Dozenten sowie auch den anderen, deren Vorlesungen nicht so stark in diese Arbeit eingeflossen sind, möchte ich daher für das mitgeteilte Wissen und manch anregenden Diskurs herzlich danken.

Ebenfalls möchte ich mich für manchen Rat und Hinweis bei Dr.-Ing. Frank Thielecke und Andreas Bernatz bedanken, die mein 2. Praxissemester – und damit auch diese Diplomarbeit – in der Abteilung mathematische Verfahren und Datentechnik des Instituts für Flugsystemtechnik des DLR vorzüglich begleitet haben.

Der gesamten Abteilung danke ich für die freundliche Aufnahme, die gute Zusammenarbeit und die nette Atmosphäre.

Ein weiterer großer Dank geht an Olaf Guth, ohne dessen Mitwirkung – und unnachahmliche Art, meine Ideen und Meinungen zu teilen, aber völlig anders auszudrücken – das dip-Framework heute längst nicht so umfangreich wäre, dass es eine solide Grundlage für weitere Forschung und Arbeit bildet.

Nicht zuletzt möchte ich meiner Familie danken. Meinen Eltern und Großeltern, ohne deren finanzielle Unterstützung mein Studium in dieser Art und Weise nie möglich gewesen wäre, sowie meiner Frau und meiner Tochter für Nachsicht, Rücksicht und jede Menge moralische Unterstützung.

Wolfenbüttel, den 17. Dezember 2004

Lukas Goormann

I. Einführung

I.1. Autonome Maschinen und Bildverarbeitung

Schon sehr lange träumen die Menschen davon, Maschinen zu entwickeln, welche die ihnen gestellten Aufgaben vollkommen selbständig, autonom erledigen. Maschinen, auf die man sich verlassen kann, so dass man sich gar nicht oder nur wenig um die übertragenen Aufgaben kümmern muss. In Computersystemen gibt es solche autonomen Helfer schon seit geraumer Zeit auf Softwareebene. Sie übernehmen immer komplexere Aufgaben. Man denke dabei z.B. an die Daemons und Dienste, die für Betriebssysteme die Verwaltung des Speichers oder das Versenden und Entgegennehmen von elektronischer Post erledigen, oder die Virenscanner, die einerseits im Hintergrund ständig die ausgeführten und gespeicherten Programme und Daten überwachen, regelmäßig die Festplatten durchforsten und sich andererseits selbständig bei bestehender Internetverbindung auf den neusten Programm- und Wissensstand bringen.

Auch „im Realen“ nimmt die Zahl der autonom arbeitenden Maschinen ständig zu. Bekanntere Beispiele hierfür sind neben den Industrierobotern, die in der Fertigung arbeiten und auf unterschiedliche Rahmenbedingungen angepasst reagieren, die Staubsauger- und Rasenmäher-Roboter. Sie können selbständig die erreichbare Bodenfläche reinigen bzw. mähen, achten dabei auf ihren Energiestand und fahren, wenn nötig, selbst zur Ladestation, um sich zu „betanken“.

Mit zunehmender Unabhängigkeit und Flexibilität dieser autonomen Maschinen und mit der Fülle der zu bewältigenden Situationen wachsen die Anforderungen an die verwendete Sensorik, das Verständnis und die Reaktionsfähigkeit in einer unter Umständen unbekannten und dynamischen Umwelt.

Ein Rasenmäher-Roboter braucht z.B. zur Zeit noch verlegte Grenzmarkierungen, welche ihm zeigen, wohin er nicht fahren darf, weil dort beispielsweise der Rasen aufhört oder ein Gartenzwerg steht. Seine Umweltwahrnehmung beschränkt sich also auf „erlaubter Bereich“, „verbotener Bereich“ und „Ladestation“. Das könnte für manche Besitzer eines solchen Rasenmäher-Roboters nicht ausreichend sein.

Falls der Besitzer beispielsweise kurz geschnittene, eintönig grüne Wiesen nicht mag und möchte, dass der Roboter alle Gänseblümchen stehen lässt, so kann er

ihm dies nicht direkt mitteilen, da der Roboter Gänseblümchen nicht wahrnehmen und folglich auch nicht erkennen (lernen) kann.

Soll der Roboter dennoch eingesetzt werden, so bliebe als einzige, absurde Möglichkeit, jedes Gänseblümchen als „verbotenen Bereich" zu kennzeichnen. Diese Kennzeichnung wäre ständig neu wachsenden und verblühenden Gänseblümchen anzupassen. Der Roboterbesitzer ist dann zwar nicht mehr mit Mähen, dafür aber mit dem Verlegen von Grenzmarkierungen beschäftigt.

Besser wäre es in diesem Fall, der Roboter könnte die Gänseblümchen oder andere verbotene Objekte und Gefahren (Gartenteich, Maulwurfshügel, ...) selbständig erkennen und meiden.

Wir Menschen und auch viele Tiere tun dieses unter anderem mit dem Sehsinn, also mit Hilfe der Augen und der dazugehörigen Informationsverarbeitung im Gehirn. Wir sehen die Gartenteiche und Gänseblümchen um uns herum und reagieren entsprechend unserer Absichten darauf. Der Sehsinn erlaubt uns, im Gegensatz beispielsweise zum Tastsinn, auch entfernte Dinge detailliert wahrzunehmen und zu bewerten, so dass wir im Voraus planen können.

Wir können Sackgassen von vorn herein meiden, Dingen ausweichen, die sich auf uns zu bewegen, Dinge suchen, finden und merken, sowie uns anhand des Gesehenem orientieren und auch Rückschlüsse auf unseren eigenen Zustand, wie die eigene Position und Bewegung ziehen.

Solche Fähigkeiten sind natürlich auch für autonome Maschinen von großem Nutzen. Zudem wird das Verständnis für ihre Funktionsweise und damit die Kommunikation mit einer sehenden Maschine erheblich erleichtert. Es ist nämlich einfacher, mit jemandem oder etwas zu kommunizieren, der bzw. das über einen ähnlichen „Erfahrungsschatz", ein ähnliches Umweltmodell verfügt. Der im Beispiel oben erwähnte Roboterbesitzer könnte einem sehenden Rasenmäher-Roboter ein Gänseblümchen beschreiben (Größe, Farben, Formen) oder ihm ein Bild von einem Gänseblümchen zeigen, anstatt umständlich Bereiche zu kennzeichnen.

Um einer Maschine das Sehen „beizubringen", ist es allerdings nicht damit getan, ihr eine oder mehrere Kameras einzubauen. Auch jene Arbeit, welche beim Menschen (hauptsächlich) das Gehirn erledigt, die Informationsverarbeitung, muss umgesetzt werden.

Diese Informationsverarbeitung gliedert sich bei Maschinen hauptsächlich in zwei Teile:

1. Die Bildverarbeitung, die aus den Kamerabildern die benötigten Informationen extrahiert, deutet und die Ergebnisse – der jeweiligen Aufgabe angepasst – zur Verfügung stellt.

2. Eine globalere Kontrollinstanz, welche die erlangten Informationen und Interpretationen in Zusammenhang mit vorher oder von anderen Sensoren gemessenen Informationen stellt, prüft und daraus Reaktionen ableitet, welche zu veränderten Anfragen an die Bildverarbeitung führen können.

Die Bildverarbeitung lässt sich wiederum in Bildaufbereitung (image preprocessing) und Bildanalyse (image analysis) untergliedern. In der Bildaufbereitung wird das Bild so vorbearbeitet, dass für die spezifische Analyse unwesentliche Informationen (z.B. kamerabedingtes Rauschen, wetterbedingte Helligkeitsschwankungen) herausgefiltert und wesentliche Inhalte (z.B. Kontraste, Kanten) verstärkt werden. In der Bildanalyse werden dann die verbliebenen und verstärkten Informationen ausgewertet und interpretiert. Bildaufbereitung und -analyse können auch zyklisch für ein Bild mehrfach durchlaufen werden, um z.B. mit neuen Parametern, die aus der Bildanalyse gewonnen wurden, die Ergebnisse zu verbessern oder zu validieren.[1]

Eine weitere wichtige Komponente beim maschinellen „Sehen“ ist die Zeit. Unter diesen Aspekt fällt hierbei nicht nur die Ausführungszeit, die für die Bildverarbeitungsalgorithmen benötigt wird, sondern auch Zeit als physikalische Größe in der beobachteten Welt, der alle Abläufe unterworfen sind. Alle von der Kamera aufgenommenen Veränderungen in der Welt brauchen eine gewisse Zeitspanne und sind innerhalb dieser kontinuierlich[2].

Für klassische Bildverarbeitungsaufgaben ist der zeitliche Aspekt zwar noch recht irrelevant, wie z.B. für das Entscheiden, ob ein produziertes Teil defekt „aussieht“ oder ob ein bestimmter Typus Pfandflasche bekannt ist und wie viel Geld dem Kunden folglich gutzuschreiben ist. Doch sie stellt für kontinuierliche

1 Diese Einteilung ist lediglich eine grobe theoretische Gliederung der Aufgaben, die im Allgemeinen von verschiedenen Teilen der verwendeten Soft- oder auch Hardware erledigt werden. In der Praxis existieren auch Module und Algorithmen die, je nach Kontext, sowohl Bildaufbereitung als auch -analyse zuzuordnen sind.

2 Die Kamera nimmt dies allerdings unter Umständen nicht wahr, da sie ja das Geschehen nur abschnittsweise wahrnimmt, die Zeit also diskretisiert.

Aufgaben, wie selbständige Fortbewegung oder Verfolgung (tracking) von Objekten, insbesondere in einer dynamischen Umgebung, eine wichtige und kritische Größe dar. Viele Algorithmen aus der klassischen Bildverarbeitung zu statischen (Einzel-)Bildern können daher nicht direkt übernommen werden, sondern müssen unter diesem Gesichtspunkt überdacht werden.

Es ergeben sich aus der Kontinuität einerseits viele Vereinfachungen (das Gesehene wird sich in hinreichend kleinen Abständen nur geringfügig verändern), andererseits aber auch spezielle Anforderungen an die Bildverarbeitung. Diese muss nämlich schnell genug in „Echtzeit“ (Realtime) ein Einzelbild bearbeiten können, um keine wesentlichen Änderungen und Informationen zu „übersehen“.

Gerade um dieser Anforderung gerecht zu werden, ist es erforderlich, dass die Bildverarbeitung – neben dem Einsatz effizienter Algorithmen – hochangepasst an die einzelne Situation nur das wirklich Notwendige tut und soviel Vorwissen wie möglich in die Parameter und Auswahl der verwendeten Verfahren integriert.

Ein gutes, robustes Bildverarbeitungssystem für hoch entwickelte autonome Maschinen ist also nicht nur schnell, adaptiv und fein parametrisierbar, sondern auch flexibel und dynamisch in der Anwendung verschiedener Algorithmen.

I.2. Ein autonomer Kleinhelikopter: Das ARTIS Projekt

Eine besondere Herausforderung ist das autonome Fliegen. Einerseits bietet die Fortbewegung in der Luft viel Freiheit und Flexibilität – autonome, fliegende Maschinen können so für eine Vielzahl von Aufgaben in Frage kommen – andererseits steigen mit der Freiheit auch die Anforderungen an Steuerung, Navigation und Umweltwahrnehmung.

ARTIS (Autonomous Rotorcraft Testbed for Intelligent Systems) ist eine Versuchsplattform, basierend auf einem modifizierten Modellhelikopter, mit der am Institut für Flugsystemtechnik (FT) des DLR (Deutsches Zentrum für Luft- und Raumfahrt) verschiedenste Techniken, Hard- und Software für den Einsatz in autonomen UAVs (Unmanned Aerial Vehicles) erforscht werden.

Abbildung I.1: ARTIS im Flugversuch

Hierzu wurde ein Navigations- und Regelungssystem für den autonomen Flug basierend auf handelsüblichen Sensoren und Komponenten realisiert.

Nach den ersten autonomen Flügen sollen verschiedene neue und alternative Lösungen implementiert und auf ihre Tauglichkeit in den unterschiedlichsten Szenarien getestet werden.

Eine besondere Rolle spielt dabei auch die Erforschung des maschinellen Sehens als multifunktionalem Sensor- und Steuerungswerkzeug.

ARTIS ist in der Basiskonfiguration mit D-GPS, 3-Achsenmagnetometer, Trägheitsplattform, Flug- und Navigationsrechner, Sonar-Höhenmesser, W-LAN und Funkmodem ausgestattet. Diese Grundausstattung kann aufgrund ihres modularen Aufbaus sehr leicht verändert und durch Nutzlast ergänzt werden.

Abbildung I.2: Die ARTIS Bodenstation

Des Weiteren gibt es eine mobile Bodenstation, von der aus die Flugmanöver usw. kommandiert und die Experimente anhand vielfältiger Anzeigen überwacht werden. Auch Änderungen, beispielsweise am Programmcode oder an den voreingestellten Parametern, können mittels dieser Bodenstation im Feld vorgenommen und auf die Bordrechner des ARTIS Helikopters eingespielt werden. Das Betriebssystem dieser Bodenstation

ist Microsoft Windows XP. Per LAN und W-LAN können weitere Rechner zur Beobachtung und Steuerung komplexer bzw. paralleler Experimente angeschlossen werden, so dass mehrere Benutzer gleichzeitig Einfluss auf den Versuchsverlauf nehmen können. [1]

I.3. Die Bildverarbeitungshardware

Für die Bildverarbeitungsexperimente wird der ARTIS Hubschrauber mit einem eigenen Bildverarbeitungscomputer und einer „Onboard"-Kamera ausgerüstet.

Der Bildverarbeitungsrechner ist aus PC 104+ Komponenten rund um einen Intel Pentium M Prozessor mit 1,4 GHz interner Taktfrequenz aufgebaut. Er ist mit 512 MB RAM einer 1 GB Flashdisk und einer FireWire-Karte (IEEE 1394) bestückt. An jene ist die DMK 21F04 von „The Imaging Source", eine 8-bit-Monochrom-Kamera mit einer Auflösung von 640 x 480 Pixel und einer 30-Hz-Bildrate angeschlossen. Als Betriebssystem für die onboard Bildverarbeitung kommt eine minimale SuSE 9.1 Linux Installation zum Einsatz.

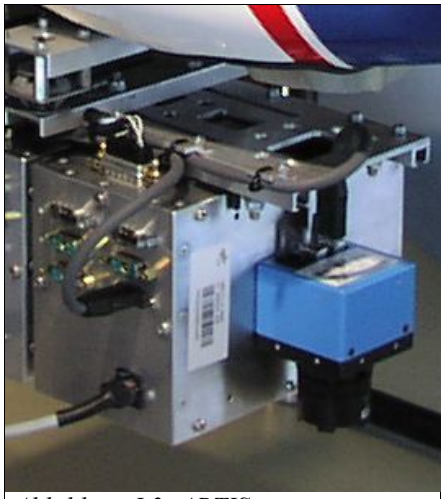


Abbildung I.3: ARTIS Bildverarbeitungshardware

I.4. Die Aufgabe: „Relatives Hovern"

Die digitale Bildverarbeitung in autonomen Helikoptern kann – neben Kollisionsvermeidung, Wegfindung u.a. – auch zum relativen Schwebeflug, dem „relativen Hovern" eingesetzt werden. Hiermit ist das Positionhalten relativ zu Teilen oder dem Gesamten des im Kamerabild Aufgenommenen gemeint. Dies kann, wie beim „normalen" Schwebeflug der Boden und damit die Welt sein, z.B. als Unterstützung oder Ersatz des Schwebeflugs mit Hilfe des GPS, insbesondere wenn dieses gestört ist.

Es ist aber auch denkbar, dass die Position relativ zu einem in der Welt bewegten Objekt, (einem Auto oder einer Person usw.), gehalten werden soll, z.B. zur Verfolgung und Beobachtung oder um einfach an den Einsatzort geführt werden zu können. Gerade der Schwebeflug relativ zu bewegten Objekten lässt sich mit den herkömmlichen Sensoren an Bord eines autonomen

Helikopters kaum bewerkstelligen. Im ARTIS Projekt wurden bereits erste Verfahren zur Realisierung eines solchen Schwebefluges relativ zu einem künstlichen, bekannten Muster untersucht. [2]

Im Rahmen dieser Diplomarbeit sollten alternative Algorithmen – auch zum Positionhalten über beliebigem, im Vorfeld unbekanntem Untergrund – erarbeitet, implementiert, getestet und bewertet werden. Dazu war in Absprache und Zusammenarbeit mit Olaf Guth, der eine weitere Diplomarbeit aus dem Themenbereich der digitalen Bildverarbeitung im ARTIS Projekt anfertigte, die notwendige Softwareumgebung zu erstellen. Gegeben war die im vorherigen Abschnitt beschriebene Hardware, wobei die Kamera relativ zum Hubschrauber nach unten gerichtet ist.[3]

II. Strategien zur Umsetzung des relativen Hoverns

II.1. Allgemeine Strategieüberlegungen

Wie in der Aufgabenstellung (Abschnitt I.4) schon beschrieben wurde, geht es beim relativen Hovern darum, die Position relativ zum von der Kamera Wahrgenommenen zu halten. D.h. alle relativen Positionsänderungen müssen ausgeglichen werden. Relative Positionsänderungen können auf zwei Arten zustande kommen:

1. Die Hubschrauberposition wird verändert. Dies ist z.B. durch Wind oder andere Einflüsse von außen, aber auch durch nicht perfekte Trimmung, Fehl- und konkurrierende[1] Eingaben des Navigationsrechners, anderer Sensoren oder auch der Bildverarbeitung und der Algorithmen zum relativen Hovern[2] möglich.

2. Das beobachtete Objekt bewegt sich. Dies ist insbesondere in den Szenarien der Fall, in denen einem Auto oder Objekt gefolgt werden soll.

Abbildung II.1: ARTIS folgt einem Auto (Simulation)

Es gibt mehrere denkbare Strategien, wie man die Positionsänderungen ausgleichen kann. Eine davon wäre, sich das Bild bzw. besonders markante Eigenschaften des Bildes im Sollzustand zu merken, ständig diese mit dem aktuellen Bild bzw. dessen Eigenschaften zu vergleichen und Rückschlüsse zu ziehen, wie der Sollzustand wieder hergestellt werden kann. Ist z.B. die Mitte des Sollbildes in eine bestimmte Richtung verschoben worden, so muss man sich in die entgegengesetzte Richtung bewegen, um dieses auszugleichen, haben sich die Größenverhältnisse proportional zueinander verändert, so scheint eine Ausgleichsbewegung nach oben bzw. unten notwendig.

---

1 z.B. zur Vermeidung von Kollisionen mit anderen Objekten

2 z.B. erzeugt durch Messungenauigkeiten oder -fehler

Der Vorteil dieser Strategie ist es, dass in jedem Schritt erneut und unabhängig vom vorherigen Schritt gemessen wird, Messfehler können sich also nicht akkumulieren, eine Drift wird vermieden. Nachteil des Verfahrens ist, dass in jedem Bild genügend Elemente des Sollbildes wieder findbar sein müssen, damit eine Beziehung zu diesem hergestellt werden kann. Außerdem muss sicher gestellt werden, dass die (gemerkten) Elemente des Sollbildes so eindeutig sind, dass sie stets fehlerfrei wieder gefunden werden können[1] und es keine Möglichkeit gibt, diese durch Drehung, Skalieren oder Verschieben in anderer Form als der Sollposition auf sich selbst abzubilden[2].

Diese Strategie eignet sich daher vor allem zum Schweben über vorgegebenen künstlichen Mustern, da man diese dementsprechend gestalten kann.

Eine weitere Vorgehensweise ist die Messung der relativen Eigenbewegung in jedem Schritt, die man dann aufsummiert. Wird der resultierende Vektor negiert, ergibt sich daraus jeweils die benötigte Korrekturbewegung. Vorteile dieses Verfahrens sind, dass einerseits lediglich in jedem Bild genügend viele Elemente des jeweils vorangegangenen wieder gefunden werden müssen, um eine Eigenbewegung zu schätzen, und dass andererseits die Schätzung der Eigenbewegung auch für andere Zwecke, z.B. Verifizierung der Messungen anderer Sensoren[3], genutzt werden kann. Allerdings wirken sich Messfehler viel länger aus, da der Korrekturvektor auf allen gemachten Messungen der Eigenbewegung beruht. Diesem kann man begegnen, indem man nicht nur von Bild zu Bild sondern auch über einen längeren Zeitraum die Eigenbewegung misst. Z.B. könnte man sich über einen längeren Zeitraum markante Punkte merken, die bereits überflogen wurden, und, wenn diese noch oder wieder sichtbar sind (bzw. sein müssten), die gemachten Messungen daran überprüfen[4].

Natürlich lassen sich auch die hier genannten Vorgehensweisen kombinieren. So ist es denkbar, dass man über einem künstlichen Muster, solange es genügend sichtbar ist, nach der ersten Strategie verfährt, sollte es aber mal aus dem Sichtbereich verschwinden, auf die zweite Strategie ausweicht.

---

1 Wenn sich die gemerkten Elemente wiederholen – inner- oder außerhalb des Sollbildes – z.B. falls der Helikopter über einem großen Schachbrett-Muster oder der Mittellinie einer Straße schwebt, so kann es viele Möglichkeiten geben, das Sollbild wiederherzustellen.

2 Ist z.B. das Sollbild ein Kreis, so kann eine Drehung um dessen Mittelpunkt nicht festgestellt werden.

3 Falls der Hubschrauber sich nicht über bewegten Objekten befindet.

4 Auch dies geht selbstverständlich nur, wenn man sich sicher sein kann, dass sich die Landschaft bzw. die gemerkten Punkte nicht verändern.

Welches Verfahren man auch verwendet, es ist in jedem Fall dafür zu sorgen, dass die verwendeten Algorithmen so schnell wie möglich sind. Gerade beim Positionshalten relativ zu in der Welt beweglichen Objekten ist dies absolut erforderlich, denn wenn diese für längere Zeit nicht von der Kamera erfasst werden, ist es ohne Zusatzinformationen quasi unmöglich, diese gezielt wieder zu finden. Zudem sind so nur kleine Kurskorrekturen nötig, und bei sehr ähnlichen aufeinander folgenden Bildern ist auch eine Fehlmessung der Eigenbewegung unwahrscheinlicher.

Für eine möglichst schnelle Bildverarbeitung sind vor allem zwei Dinge wichtig:

1. Vorwissen integrieren. Dies führt in der Regel zu einfacheren, angepassten Algorithmen.

2. Schnelle Reduktion der Datenmenge und Verdichten, Herausfiltern der benötigten Informationen innerhalb der Algorithmen, denn je größer die Menge der zu bearbeitenden Daten in einem Schritt des Algorithmus ist, desto länger dauert er.

Letzteres kann man z.B. durch Konzentration auf wenige, markante Details erreichen. So merken sich viele Algorithmen Eck- und Kreuzungspunkte von Kanten (corners) und versuchen, diese wieder zu finden. Da Verfahren dieser Art schon vielfältigst untersucht worden sind [4][5], behandelt diese Arbeit vorrangig das Verdichten und Filtern der Information durch Konzentration auf besonders markante (z.B. helle oder dunkle) Bereiche, Flecken (Blobs). Diese haben den Vorteil, dass sie selbst jeweils unterschiedliche Eigenschaften haben, welche sich der Algorithmus merken kann, und dass sie daher leicht wieder zu finden und schnell zu identifizieren sind.

II.2. Hovern über vorgegebenen künstlichen Mustern

Im Prinzip ist das Hovern über vorgegebenen künstlichen Mustern[1] ein Spezialfall des Hoverns an spezifischen Stellen. Das heißt, ein Algorithmus, der ein Hovern an beliebigen Stellen ermöglicht, sollte auch beim Hovern über einem solchen Muster funktionieren, und sogar besonders gut, wenn das Muster auf

1 Mit „vorgegebenen künstlichen Mustern“ sind in diesem Abschnitt speziell für das Hovern entworfene Muster gemeint, im Gegensatz zu beliebigen Untergründen und ähnlichem, welche natürlich auch menschlichen Ursprungs, insofern also ebenso künstlich seien können.

jenen Algorithmus abgestimmt wurde. Doch ein bekanntes künstliches Muster bedeutet auch Vorwissen, auf das man die Algorithmen anpassen kann, um z.B. Geschwindigkeitsvorteile zu erlangen. Falls die Maße des Musters bekannt sind und es hinreichend viele markante Punkte bietet, so sind sogar Rückschlüsse auf die eigene Position relativ zum Muster in jedem einzelnen Bild möglich, soweit das Muster zu einem genügenden Teil von der Kamera erfasst wird.

Da solch ein spezialisierter Algorithmus „nur" das entsprechende Muster suchen und wieder finden wird, können sich die anderen Bildelemente auch beliebig ändern, ohne dass dies Einfluss auf das Hover-Verhalten des Helikopters hat. Das ist insbesondere beim Hovern relativ zu bewegten Objekten von Vorteil.

Künstliche Muster wurden aufgrund dieser Eigenschaften auch schon des Öfteren eingesetzt, um Landeplätze für autonome Hubschrauber zu markieren und den Landevorgang mit Hilfe von visuellen Sensoren zu unterstützen.[6]

Wichtig ist allerdings, dass der verwendete Algorithmus in der Lage ist, nicht nur ein existierendes Musters in einem Bild korrekt zu erkennen, sondern auch das Fehlen des Musters. In solchen Fällen könnte dann z.B. das Hovern abgebrochen oder auf einen anderen weniger spezialisierten Hover-Algorithmus umgeschaltet werden. Schlecht wäre es hingegen, wenn das Muster an Bildstellen „gefunden" wird, wo es sich nicht in der Realität befindet.

Um dies zu vermeiden, sollte also

1. das Muster sehr unterschiedlich von der Einsatzumgebung sein und

2. der eingesetzte Algorithmus auch diese eindeutigen Unterschiede feststellen und überprüfen können.

Ein gutes Muster sollte zudem, wie oben (Abschnitt II.1) schon erwähnt, nicht durch Verschieben, Drehen und Skalieren[1] auf sich selbst abgebildet werden können.

1 Ausnahme sind natürlich Drehungen um Vielfache von 360°, Verschiebungen um den Null-Vektor und Skalierungen um 1.

II.3. Hovern an beliebigen Stellen

Da die Kamera die Welt als Projektion auf eine (zweidimensionale) Ebene wahrnimmt, können Rückschlüsse auf die eigene (relative) Position und Lage im (dreidimensionalen) Raum nur mittels Zusatzinformationen gewonnen werden.

Diese Zusatzinformationen können, wie oben (Abschnitt II.2) erwähnt, die realen Maße von Gesehenem sein, dann ist es möglich[1] die relative Position und Lage direkt, mit Hilfe einfacher Berechnungen (Strahlensatz), aus einem Bild abzulesen. Als Zusatzinformationen können aber auch die Messungen anderer Bordsensoren und die eigene Position und Lage (relativ zum Gesehenen) zum Zeitpunkt der letzten Messung sowie das dazu gehörige Bild (falls dieses genügend Überschneidung bzw. Ähnlichkeit mit dem jetzigen aufweist) dienen.

Jede dieser Messungen ist grundsätzlich fehlerbehaftet, z.B. dadurch, dass eine digitale Kamera nur eine bestimmte diskrete Auflösung besitzt. Man kann aber versuchen, diese Fehler über die Zeit mit Hilfe von Heuristik und Wahrscheinlichkeitsberechnungen zu minimieren. Wenn z.B. das Abbild eines Punktes in Pixel A lokalisiert wird und zwei Bilder später im benachbarten Pixel B, so ist es erfahrungsgemäß wahrscheinlich, dass es sich im Zwischenbild eigentlich zwischen den Pixeln befand. Genauso ist es erfahrungsgemäß wahrscheinlich, dass sich Bilder und Zustände wie z.B. die Geschwindigkeit von Objekten in sehr kurzen Zeitabschnitten nur wenig ändern. Daraus folgernd kann ein Algorithmus Ergebnisse qualifizieren, sie bewerten[2]. So kann dann in Fällen, in denen ein besonders unwahrscheinliches Ergebnis herauskam, dieses nochmals mit Hilfe eines anderen Verfahrens oder anderer Sensordaten validiert werden, oder eine Entscheidung, eine Reaktion bis zur Überprüfung mit Daten aus dem nächsten Bild verschoben werden.

Wenn Vorkenntnisse von der Beschaffenheit des Untergrundes vorhanden sind, lassen sich natürlich auch hier die Algorithmen anpassen. So ist das Hovern über einer Ebene wesentlich einfacher zu realisieren als in einer unebenen Landschaft. Bei einer Ebene muss man nur den Abstand zu einem Punkt in dieser kennen. Bei unebenem Untergrund muss man hingegen den Abstand zu jedem

1 Sobald z.B. mindestens drei Punkte, deren Lage zueinander bekannt ist, identifiziert werden können.

2 „Ich habe dies und das heraus bekommen und bin mir so und so sicher, dass dieses Ergebnis stimmt."

einzelnen betrachteten Punkt wissen und ihn für neue Punkte aufgrund ihrer Lageänderung im Verhältnis zu bekannten Punkten berechnen.

Zur Schätzung der Eigenbewegung mit Hilfe von Punkten – die durch Bildverarbeitung in aufeinander folgenden Bildern als Abbild des jeweils selben Punktes in der Welt identifiziert wurden – sind schon unterschiedliche Verfahren vorgestellt worden [10][11][21]. Für die Implementierung dieser und weiterer Algorithmen wurden im eigens erstellten Bildverarbeitungsframework dip (siehe Kapitel V „Das dip-Framework“) entsprechende Schnittstellen bereit gestellt und erste Verfahren, die unterschiedliche Rahmenbedingungen berücksichtigen bzw. voraussetzen, zu Testzwecken implementiert (siehe auch Kapitel IV „Eigenbewegung“).

II.4. Interaktion mit dem Navigationsrechner zum Hovern

Um mit dem Navigationsrechner zu kommunizieren, ihm die gewünschten Flugrichtungen und Reaktionen zu den gemessenen Daten mitzuteilen, gibt es viele denkbare Wege.

So könnte man ihm absolute, streckenbasierte Befehle geben, z.B. „x Meter nach links“. Eine andere Möglichkeit wären motoren- bzw. beschleunigungsbasierte Befehle wie z.B. „Volle Kraft voraus!“. Diese müssten unter Umständen angepasst oder korrigiert werden können, falls sich die Bedingungen vor Ende der Ausführung[1] ändern. Außerdem müssen bei der Generierung dieser Kommandos die Reaktionszeit des Hubschrauber und der Navigationslösung sowie die mögliche Genauigkeit der Ausführung berücksichtigt werden.[2]

All dies ist für eine komplette, funktionsfähige Implementierung des relativen Hoverns natürlich von Bedeutung. Da die Navigationslösung, die Software für den Navigationsrechner und insbesondere die zu nutzenden Schnittstellen noch nicht zu hundert Prozent fertig bzw. einsatzbereit[3] sind, die Befehle und die Algorithmen zur Entscheidung und Generierung der Befehle aber sehr stark von hiervon abhängen, wurde in dieser Arbeit auf umfassendere Betrachtung und Diskussion jener verzichtet.

1 Zielpunkt oder gewünschte Drehzahl usw. noch nicht erreicht.

2 So macht es z.B. keinen Sinn dem Navigationscomputer zum genauen Erreichen der Zielposition zu befehlen „4 cm nach hinten“, wenn dieser die Kommandos nur auf 10 cm genau ausführen kann (Gefahr der „Verschlimmbesserung“).

3 Sie sind noch im Experimentalstadium.

II.5. Auswahl des Bildverarbeitungsverfahrens

Die meisten existierenden bzw. in der Literatur vorgestellten Algorithmen zur Messung der Eigenbewegung, beruhen auf Stützstellen, Punkten im aktuellen Bild, zu denen die Bildbewegung, bzw. der korrespondierende Punkt im Vorgängerbild bekannt sind. Die Aufgabe der Bildverarbeitung ist es, diese Punkte und die zugehörigen Bewegungen zu finden und bereit zu stellen. [10][11][21]

Wie schon in der Einführung (Abschnitt I.1) und auch in diesem Kapitel(insbesondere Abschnitt II.1) angesprochen, ist die Verarbeitungsgeschwindigkeit hierbei besonders wichtig. Ein Grund, warum viele Bildverarbeitungsalgorithmen relativ langsam sind, ist die große Menge an Daten die sie bearbeiten müssen. Ein 640x480 Pixel großes Bild enthält 307.200 Bildpunkte, jeder mit seinem spezifischen Wert. Der Wert eines Pixels allein reicht aber meist nicht aus, um ihn im gesamten Bild wieder zu finden[1]. Daher wird in einigen Ansätzen der jeweilige Pixel im Zusammenhang mit seiner näheren Umgebung wieder gesucht. Je nach Implementierung können dafür sehr viele Pixel-Vergleiche notwendig sein. In anderen Ansätzen werden Pixel nur in einer vorgegebenen lokalen Nachbarschaft wiedergesucht, dass lässt die Anzahl der Vergleiche zwar sinken, aber Bewegungen, die über den Suchbereich hinausgehen, können nicht wahrgenommen werden[2].

Die Idee war daher, eine alternative Vorgehensweise zu finden, bei der zunächst die Datenmenge stark reduziert wird, dabei relevante Informationen konzentriert, unwichtige Details herausgefiltert werden, und diese Informationen und Ergebnisse dann so zu speichern, dass sie besser identifizierbar und direkt vergleichbar sind. Hierzu erschien das Konzept der Objektorientierung besonders hilfreich, da klassifizierte Objekte, die mehrere spezifische Attribute haben, leichter und schneller zu vergleichen sind, als primitive Datenstrukturen wie Arrays, Reihen von Pixelwerten, in denen die für die Identifizierung benötigten Zusammenhänge, Muster usw. stets für jeden Vergleich erneut ermittelt werden müssten. Außerdem kann man mit Hilfe der Methoden, die zu einer Klasse von Objekten zur Verfügung stehen, diese Objekte ändern und auch einfach und effi-

1 Dies gilt vor allem, zumal sich der Pixelwert auch leicht ändern kann, aufgrund vieler Einflüsse wie z.B. Änderung der Lichtverhältnisse.

2 Diesen Problemen wird manchmal mit einer so genannten Bild-Pyramide begegnet, diese muss dann aber wiederum Bild für Bild erstellt werden und kann trotzdem nicht immer bei großen Bewegungen eine Falschidentifikation (ein „Durchrutschen“ der Punkte) vermeiden. [4]

zient auf weitere abgeleitete Eigenschaften zugreifen, ohne dass sich etwas an der Vergleichbarkeit ändert.

Für diese Arbeit wurde eine solche alternative Vorgehensweise mit Hilfe der „Blob-Analyse“ (vorgestellt im nächsten Kapitel) realisiert und untersucht, ob sich so ermittelte Punkte zur Schätzung der Eigenbewegung – und damit zum „relativen Hovern“ – eignen und ob dieser Ansatz tatsächlich das Potential bietet, diese Aufgabe schneller zu erfüllen als andere bisher verwendete Algorithmen.

III. Blob-Analyse

III.1. Blobs vorgestellt

„Blob“ heißt aus dem Englischen übersetzt soviel wie Klecks oder auch Klumpen. Umgangssprachlich kann das Wort auch verwendet werden, um Dinge, die man nicht im Detail erkennen kann, z.B. weil sie weit entfernt sind, zu beschreiben. [12]

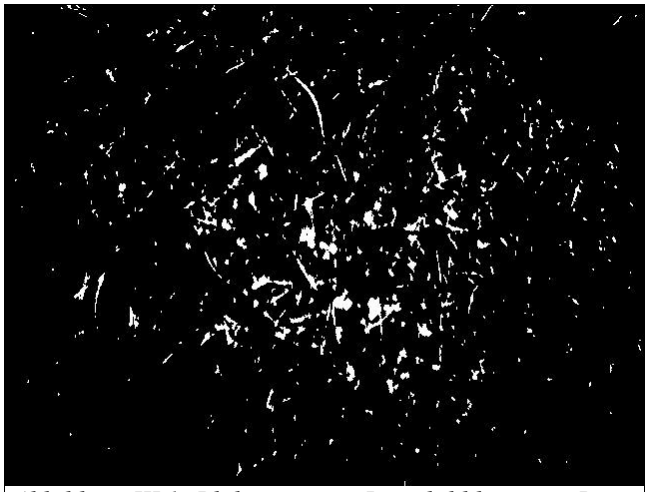


Abbildung III.1: Blobs in einem Binärbild können z.B. alle zusammenhängenden weißen Flächen sein

In der Bildverarbeitung ist mit „Blob“ immer eine zusammenhängende Menge von Pixeln gemeint. Sie haben eine oder mehrere Eigenschaften, die sie verbinden bzw. zusammen gehörig erscheinen lassen, wie z.B. einen ähnlichen Farbwert. Zusätzlich hat jeder Blob weitere Merkmale, mit denen er sich beschreiben lässt. Das kann die Pixelanzahl sein, die Form, der Umfang oder auch die Position im Bild. In der Bildverarbeitung wurden Blobs bisher meist zur Mustererkennung oder zur Objektverfolgung (tracking) in Bildern einer stationären Kamera[1] eingesetzt.

Anhand ihrer Merkmale lassen sich Blobs identifizieren bzw. Entscheidungen fällen, ob auf einem Bild z.B. ein bestimmtes Muster oder ein Gesicht zu sehen ist und wo es sich befindet. Solche Entscheidungen können durch Abwägen, Vergleichen der verschiedenen Charakteristika mit den Sollwerten, aber auch Vergleichen der Werte zueinander durch Ausnutzen von Redundanzen und Abhängigkeiten, qualifiziert werden. D.h. ein Algorithmus kann beispielsweise durch Ausgabe eines Zahlenwertes mitteilen, wie „sicher“ es ist, dass ein bestimmter Blob bzw. etwas (das gesuchte Muster, Gesicht,...) mittels eines oder mehrerer Blobs in einem Bild (wieder) gefunden wurde. Dies funktioniert um so besser, je mehr Eigenschaften eines Blobs untersucht werden.

1 z.B. zur Verkehrsbeobachtung. Hauptaugenmerk liegt hierbei allerdings darauf, dass sich Dinge vor einem stationären Hintergrund bewegen.[7][8]

III.2. Die Speicherung von Blobs

Blobs können auf verschiedene Art und Weise im Speicher des Rechners abgelegt werden. Eine Möglichkeit wäre es, sich alle zu dem Blob gehörigen Pixel und ihre Werte zu merken. Je nach Größe des Blobs können das aber große Datenmengen werden, außerdem müssten zum Vergleichen von Blobs die Eigenschaften jedes mal neu extrahiert werden. Vorteilhafter ist es, wenn man sich nur die eigentlich benötigten Merkmale, wie z.B. Schwerpunkt, Pixelanzahl, Maße, Durchschnittshelligkeit usw. des jeweiligen Blobs merkt. Auf diese kann dann bei Bedarf direkt zugegriffen werden, sie müssen für jeden Blob in jedem Bild nur einmal ermittelt werden.

Wenn man bei der Auswahl der in der konkreten Implementierung gespeicherten Attribute eines Blobs darauf achtet, dass es möglich ist – ohne erneutes Betrachten aller schon erfassten zugehörigen Pixel – einen weiteren Pixel dem Blob zugehörig zu erklären, bzw. zwei vorhandene Blobs zu einem zu vereinigen, so lassen sich schnelle Algorithmen zum Extrahieren solcher Blobs aus den gegebenen Bildern finden, deren Komplexität nur linear von der Anzahl der im Bild enthaltenen Pixel abhängt.[1]

III.3. Ablauf der Blob-Analyse

Bisher wurden Blobs zur Bestimmung der Eigenbewegung von einem kameratragenden Objekt kaum verwendet. Das mag daran liegen, dass in den gängigen Algorithmen zur Bestimmung der Eigenbewegung aus Kamerabildern, neben den in Abschnitt II.3 erwähnten Zusatzinformationen, hauptsächlich die Informationen über die alten und neuen Positionen von einzelnen Bildpunkten genutzt werden.

Positionsinformationen gibt es auch bei Blobs. Wenn man einen markanten Punkt des Blobs als dessen „Position“ definiert, so lassen sich dieselben gängigen Algorithmen auch hier anwenden. Es wurde daher (zunächst einmal) die Blob-Analyse von der Eigenbewegungsschätzung[2] getrennt. In den hier imple-

1 Siehe ebenfalls Abschnitt III.4 „Algorithmen zum „Sammeln“ der Blobs“. Im Bildverarbeitungsframework zum ARTIS Projekt werden Blobs durch die Klasse `dip::blob::cl_Blob` repräsentiert. Hierzu siehe auch Abschnitt V.4 „Gespeicherte Blobeigenschaften“

2 Erläutert in Kapitel IV „Eigenbewegung“

mentierten Algorithmen wurde der Schwerpunkt als „Position“ gewählt, weil dieser sehr stabil ist.

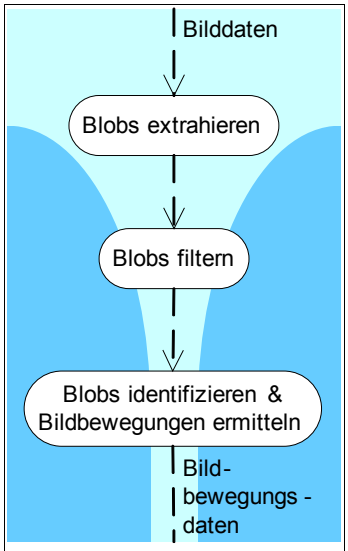


Abbildung III.2: Ablauf der Blob-Analyse

In dem vorgestellten Blob-Analyse-Verfahren[1] werden die Blobs zunächst aus einem Eingabebild extrahiert, „gesammelt“. Dann werden überzählige und schlecht identifizierbare Blobs durch Anwendung mehrerer Filter aussortiert. Die zum Schluss verbleibenden Blobs werden identifiziert und die abgeleiteten Bildbewegungen an die Algorithmen zur Schätzung der Eigenbewegung übergeben.

Man kann die Definitionen variieren, nach denen die Blobs gebildet werden. Beispielsweise können all jene benachbarten Pixel einen Blob bilden, die einen festgelegten Farbwert haben. Eine andere Regel könnte sein: Ein Pixel, der einen ähnlichen Farbwert hat wie der Durchschnittsfarbwert eines benachbarten Blobs, gehört zu diesem Blob. Je nach Regel sind unterschiedliche Algorithmen zum Sammeln der Blobs aus den Bildern nötig. Es werden zwei (einander sehr ähnliche) Blobsammelalgorithmen im nächsten Abschnitt (III.4) näher behandelt. In der vorliegenden Implementierung kann der Benutzer aus diesen beiden den bevorzugten Blobsammelalgorithmus zur Blob-Analyse selbst wählen.

Je nach eingesetztem Algorithmus zum Sammeln von Blobs und je nach Beschaffenheit des zu betrachtenden Bildes kann die Menge der Blobs sehr groß (10.000 und mehr) werden. Zwar wird die Schätzung der Eigenbewegung genauer bei einer größeren Menge von möglichst korrekten Ausgangsdaten, aber erstens lassen sich einige Blobs nicht so genau wieder finden wie andere[2] und zweitens steigt die Zahl der benötigten Schritte zur Identifizierung der Blobs schlimmstenfalls quadratisch mit der Anzahl der Blobs[3]. Es kann also sinnvoll sein die Menge der Blobs durch „Blobfilter“ zu reduzieren. Die Art und Zahl der geeigneten Blobfilter sowie deren Reihenfolge kann je nach Situation und Vorwissen unterschiedlich sein. Es wurden deshalb mehrere Blobfilter ein-

1 Implementiert in der Klasse `dip::blob::cl_BlobAnalyse`, siehe Abschnitt V.6.

2 Wenn mehrere Blobs sehr ähnliche oder gleiche Eigenschaftswerte haben, sind sie nur schwer eindeutig zu identifizieren. Dies ist insbesondere bei kleinen Blobs der Fall.

3 Genauer gesagt ist die Komplexität (O) des verwendeten Identifizierungsalgorithmus bei einem Aufruf (i) abhängig von der Anzahl der übergebenen Blobs in diesem Aufruf (n_i) und der Anzahl der Blobs vom vorherigen Aufruf (n_{i-1}). Für sie gilt schlimmstenfalls: $O(n_{i-1}, n_i) = n_{i-1} \cdot n_i$. Siehe auch Abschnitt III.6„Blob-Identifizierung“.

geführt[1], deren Parametrisierung, Anordnung, verwendete Sorten und Anzahl der Benutzer jederzeit bestimmen und ändern kann.

Eine Beispielskonfiguration zur Schätzung der Eigenbewegung mit Hilfe der Blob-Analyse wird in Abschnitt V.9 näher erläutert.

III.4. Algorithmen zum „Sammeln“ der Blobs

Die Algorithmen zum Sammeln von Blobs[2] erhalten ein Bild und erstellen eine Liste der enthaltenen Blobs.[3] Zur Zeit gibt es zwei implementierte Verfahren, die jeweils auf einer unterschiedlichen Definition von zusammengehörigen Pixeln beruhen. Das ist zum einen ein Verfahren für alle Graustufenbilder, das benachbarte Pixel, die keine (zu steile) Kante bilden als zusammen gehörig ansieht, es ist im folgenden Abschnitt III.4.1 beschrieben. Der zweite implementierte und vorgestellte Algorithmus ist für Binärbilder optimiert. Er und ein Filter bzw. ein Verfahren zum Erstellen solcher Binärbilder werden in Abschnitt III.4.2 vorgestellt.

III.4.1. Blobs aus Graustufenbildern

Der nun zunächst vorgestellte Algorithmus[4] sammelt Blobs aus Graustufen-Bildern. Die Regel, nach der er die Pixel den unterschiedlichen Blobs zuordnet, lautet: Wenn der Betrag der Differenz der Werte zweier benachbarter[5] Pixel kleiner ist als ein Grenzwert, so gehören sie zu demselben, ansonsten zu verschiedenen Blobs. Der Grenzwert kann vom Benutzer festgelegt werden. Es werden also alle Pixel des Bildes auf Blobs aufgeteilt, bzw. gibt es nach Ausführung des Algorithmus keinen Pixel, der zu keinem Blob und auch keinen Pixel der zu mehreren Blobs gehört. Der gewählte Algorithmus zur Zuordnung der

1 Näheres siehe Abschnitt III.5 „Filter für Blobs“

2 Ihre Implementierungen wurden unabhängig von den anderen Bestandteilen des Blob-Analyse-Algorithmus gehalten, so dass sie für andere Algorithmen auf Blob-Basis in jedem Fall wiederverwendet werden können.

3 Genauer gesagt erhalten sie auch die Liste der Blobs, die sie zunächst leeren und dann erneut füllen.

4 Die Klasse, in der er implementiert ist, heißt `dip::blob::cl_SimpleBlobCollector`.

5 Betrachtet wurde hier die Vierer-Nachbarschaft eines Pixels, d.h. als benachbart gilt der Pixel direkt über, unter, rechts oder links von dem betrachteten Pixel.

Pixel zu den Blobs hat den Vorteil, dass seine Komplexität nur linear von der Anzahl der Pixel des Ausgangsbildes abhängig ist. Er funktioniert im Prinzip so:

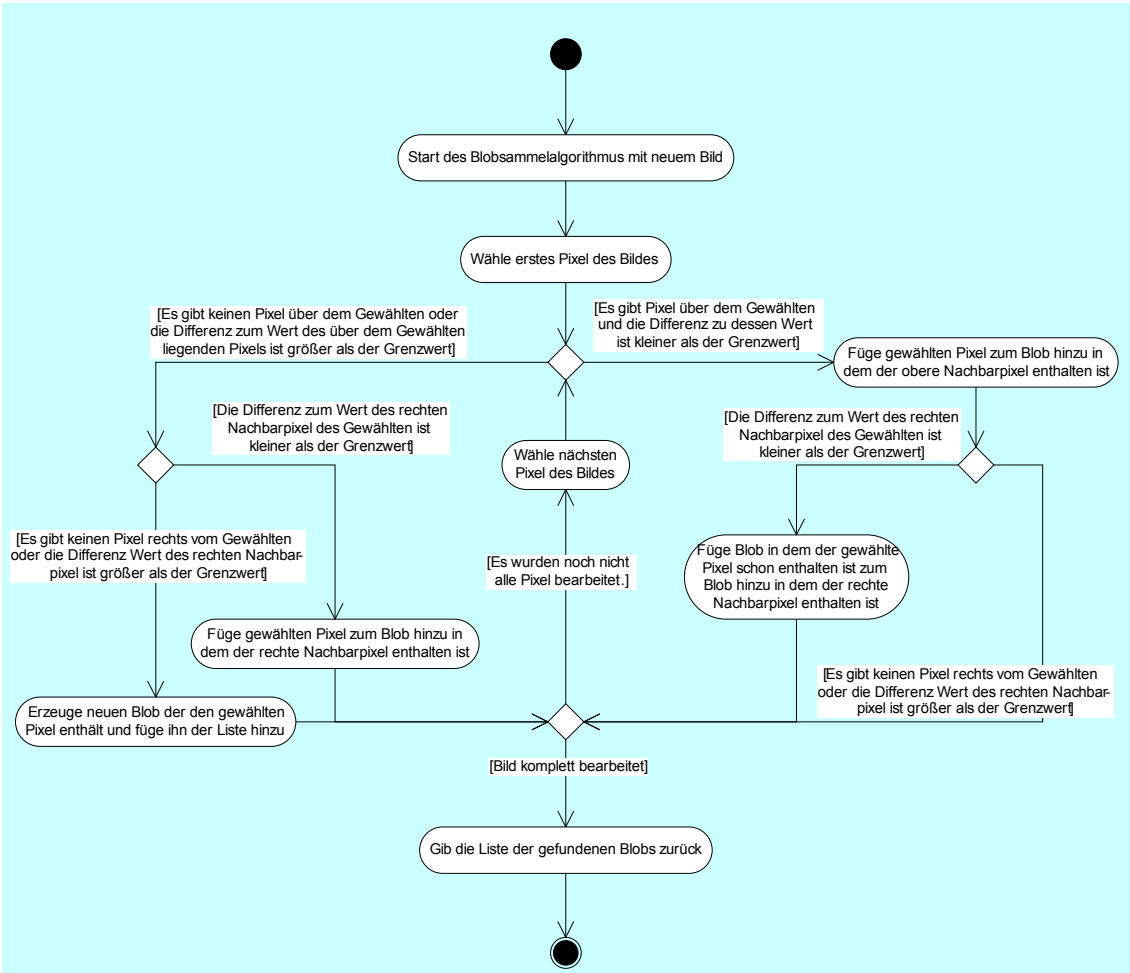


Abbildung III.3: Algorithmus zum Sammeln von Blobs aus Graustufenbildern

Die Pixel des Eingangsbildes werden pro Zeile von links nach rechts und die Zeilen von oben nach unten durchgegangen[1]. Dabei wird für jeden Pixel geprüft:

1. ob der Betrag von der Differenz der Werte des Pixels und des darüber Liegenden kleiner ist als der Grenzwert.

2. ob der Betrag von der Differenz der Werte des Pixels und des links von ihm Liegenden kleiner ist als der Grenzwert.

1 Das entspricht auch der Reihenfolge in der die Pixelwerte üblicherweise im Speicher abgelegt sind.

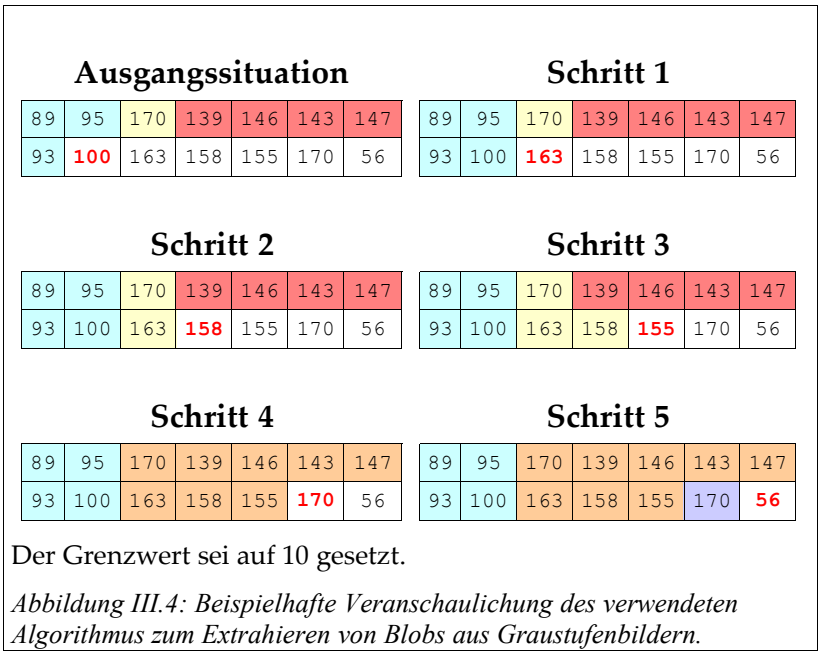
Wird 1. und 2. verneint, so wird der Pixel zu einem neuen Blob.
(Schritt 5 in Abb. III.4)

Trifft nur 1. zu, so wird der Pixel dem Blob hinzugefügt, dem auch der Pixel über ihm angehört (die Pixel links und oberhalb von ihm sind ja schon in Blobs enthalten). (Schritt 2 in Abb. III.4)

Trifft nur 2. zu, so wird der Pixel dem Blob hinzugefügt, dem auch der Pixel links von ihm angehört. (Schritt 3 in Abb. III.4)

Ist 1. und 2. wahr, so werden die Blobs, denen die Pixel links und oberhalb des betrachteten Pixel angehören, vereinigt (falls sie nicht schon ein und derselbe Blob sind), und der betrachtete Pixel wird dem resultierenden Blob hinzugefügt. (Schritte 1 und 4 in Abb. III.4)

In Abbildung III.4 ist der Algorithmus an einem Beispiel veranschaulicht. Jedes Kästchen stellt dabei einen Pixel dar, die Zahl in einem Kästchen sei der Pixelwert. Die Kästchen zugeordneter Pixel haben einen farbigen Hintergrund, gleiche Farben bedeuten dabei, dass die Pixel im selben Blob enthalten sind. Der Wert des aktuell betrachteten Pixels ist jeweils **rot** geschrieben.



Der Grenzwert sei auf 10 gesetzt.

Abbildung III.4: Beispielhafte Veranschaulichung des verwendeten Algorithmus zum Extrahieren von Blobs aus Graustufenbildern.

III.4.2. Blobs aus Binärbildern

Beim häufigen Betrachten mehrerer zu Testzwecken aufgenommener Bilderserien fiel auf, dass das menschliche Auge vielfach besonders helle oder dunkle Flecken verfolgt[1]. Diese schienen dem Betrachter auch besonders „stabil" zu sein, während Flecken mittlerer Helligkeit leichter „verschwammen" oder bei ihnen die Helligkeit eher variierte. Aus dieser Beobachtung wurde die Idee geboren, sich auch bei der Bildanalyse auf die besonders hellen oder dunklen Flecken zu konzentrieren.

Dazu wurde ein Filter implementiert, der aus einem Graustufenbild ein Binärbild erzeugt, welches besonders helle oder dunkle Pixel „markiert". Dies geschieht mit Hilfe von zwei Grenzwerten. Ein Pixel, dessen Wert zwischen den zwei Grenzwerten liegt, der also eine mittlere Helligkeit hat, wird im Ausgabebild standardmäßig[2] auf den Wert Null (schwarz) gesetzt, besonders helle oder dunkle Pixel, Pixel deren Wert entweder größer oder kleiner als beide Grenzwerte ist, werden im Ausgabebild standardmäßig auf den Wert 255 (weiß) gesetzt.

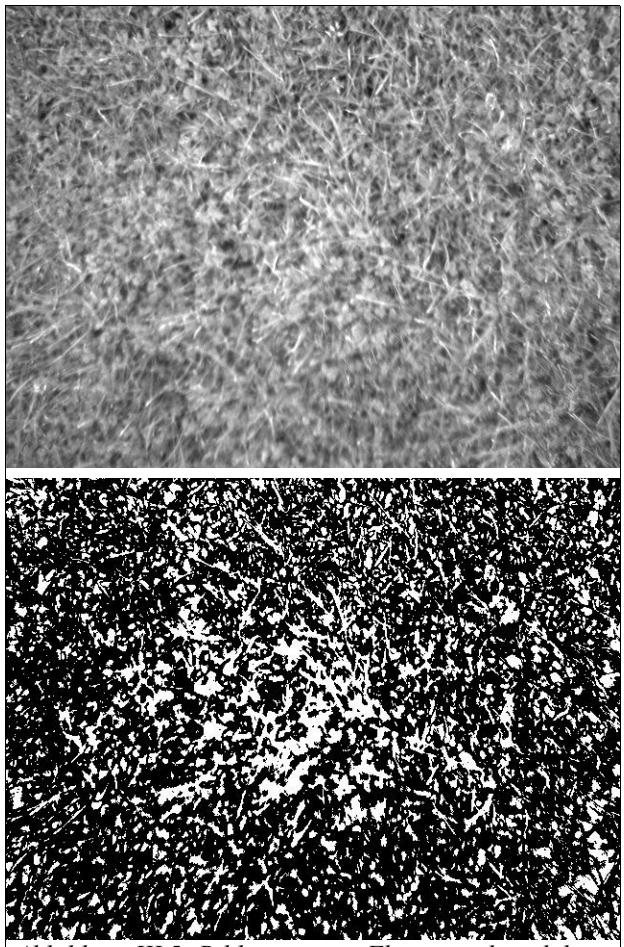


Abbildung III.5: Bild aus einem Flugversuch und das daraus mit Hilfe zweier Thresholds erzeugte Binärbild

Die beiden Grenzen des Filters können vom Benutzer jeweils auf einen absoluten Pixelwert festgelegt werden. Der Filter bietet aber auch die Möglichkeit, die Grenzwerte stets relativ zur durchschnittlichen Helligkeit des vorherigen Bildes[3] zu setzen. Das

1 Das war insbesondere dann der Fall, wenn keine konkreten Dinge zu erkennen waren.

2 Der Benutzer kann die zwei Werte im Ausgabebild seinen Wünschen anpassen. Diese Möglichkeit wird aber in den hier vorgestellten Verfahren und Beispielen nicht genutzt.

3 Den durchschnittlichen Helligkeitswert des Bildes kann man während der Anwendung der Grenzwerte ermitteln. Nimmt man also den Helligkeitswert des Vorgängerbildes, so müssen die Pixel eines Bildes in jedem Schritt nur einmal durchgegangen werden.

hat den Vorteil, dass sich der Filter an Veränderungen z.B. der Lichtverhältnisse[1] oder auch des Untergrundes[2] anpasst.

Um Blobs in solchen Binärbildern zu ermitteln, wurde ein spezieller Blobsammelalgorithmus erstellt und implementiert. Dieser Algorithmus geht von folgender Regel aus: Schwarze Pixel (mit dem Wert Null) gehören nicht zu Blobs, andere Pixel (normalerweise weiße mit dem Wert 255) gehören jeweils zum selben Blob wie die nicht-schwarzen Pixel in ihrer (Vierer-)Nachbarschaft.

Der verwendete Algorithmus funktioniert ähnlich dem im vorherigen Abschnitt (III.4.1) Beschriebenen:

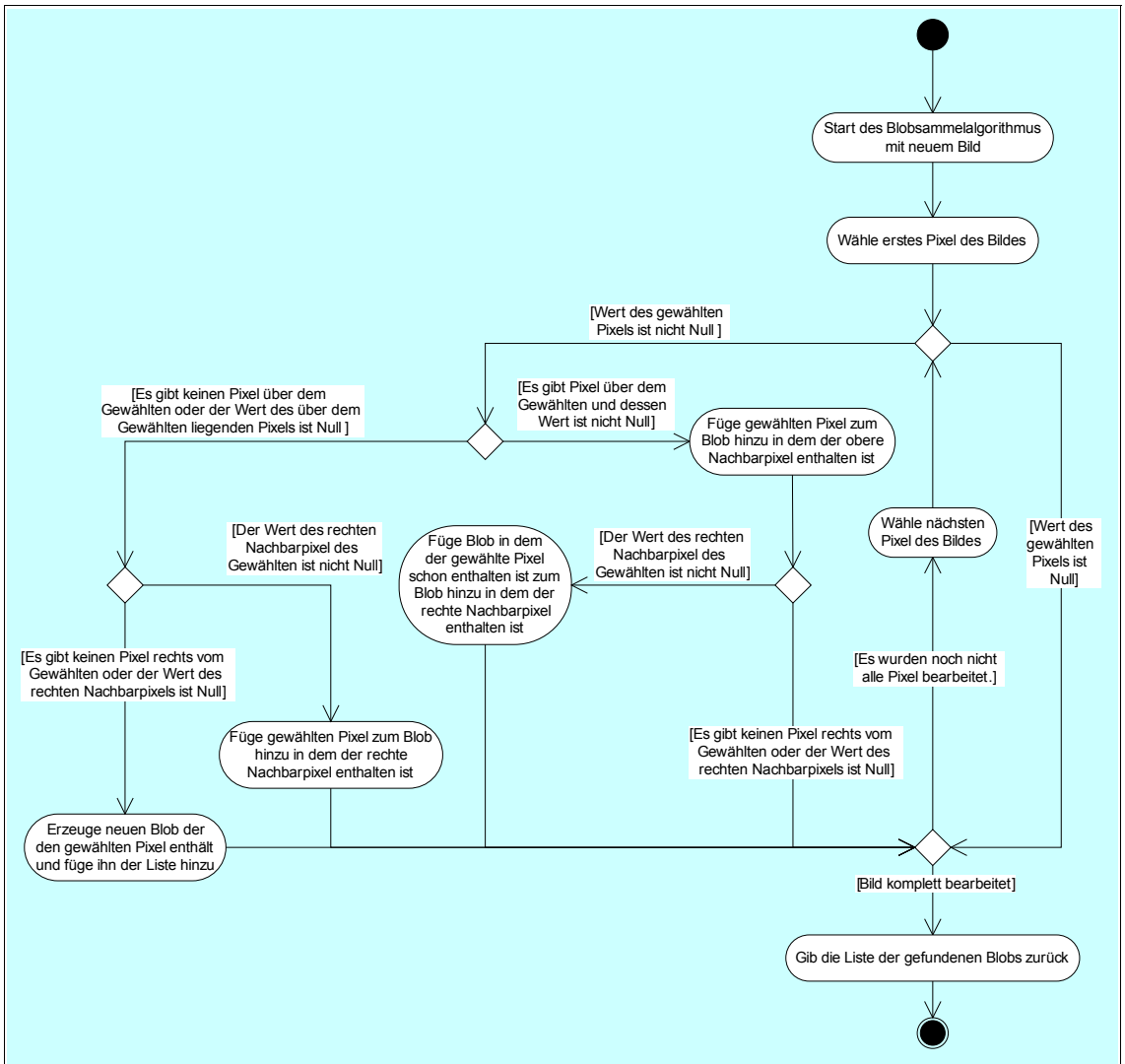


Abbildung III.6: Algorithmus zum Sammeln von Blobs aus Binärbildern

1 wenn sich beispielsweise eine Wolke vor die Sonne schiebt
2 z.B. erst dunkles Wasser, dann heller Strand

Die Pixel des Eingangsbildes werden pro Zeile von links nach rechts und die Zeilen von oben nach unten durchgegangen. Ist der Wert eines Pixels nicht Null so wird geprüft:

1. ob der Wert des über ihm liegenden Pixel Null ist

2. ob der Pixelwert zu seiner Linken Null ist

Ist 1. und 2. wahr, so wird der Pixel zu einem neuen Blob.

Trifft nur 2. zu, so wird der Pixel dem Blob hinzugefügt, dem auch der Pixel über ihm angehört.

Trifft nur 1. zu, so wird der Pixel dem Blob hinzugefügt, dem auch der Pixel links von ihm angehört.

Wird 1. und 2. verneint, so werden, die Blobs, denen die Pixel links und oberhalb des betrachteten Pixel angehören, vereinigt, falls sie nicht schon ein und derselbe Blob sind, und der betrachtete Pixel dem resultierenden Blob hinzugefügt.

Dieser Algorithmus ist – insbesondere bei Bildern mit vielen schwarzen Pixeln – um einiges schneller als der zuvor beschriebene, da nur bei nicht-schwarzen Pixeln eine eingehende Prüfung erfolgen und stets nur auf Null geprüft werden muss, aber nicht der Betrag einer Differenz mit einem Wert zu vergleichen ist.

III.5. Filter für Blobs

Die Blob-Filter werden vor allem zur Reduzierung der Anzahl zu betrachtender Blobs verwendet, damit nicht zu viele Blobs in jedem Schritt identifiziert werden müssen, so dass Bearbeitungszeit gespart wird. Hierbei ist es natürlich besonders wünschenswert, dass die verbleibenden Blobs über mehrere Bilder stabil und möglichst einfach und eindeutig identifizierbar sind. Andererseits sollen auch diese Filteralgorithmen möglichst schnell sein. Im Folgenden werden die vorhandenen Blob-Filter kurz vorgestellt.

Eine Art Blob-Filter dient dazu, Blobs mit bestimmten durchschnittlichen Pixelwerten auszusortieren. Der Benutzer kann einen Wertebereich (durch eine obere und eine untere Schranke) angeben und bestimmen, ob Blobs deren durch-

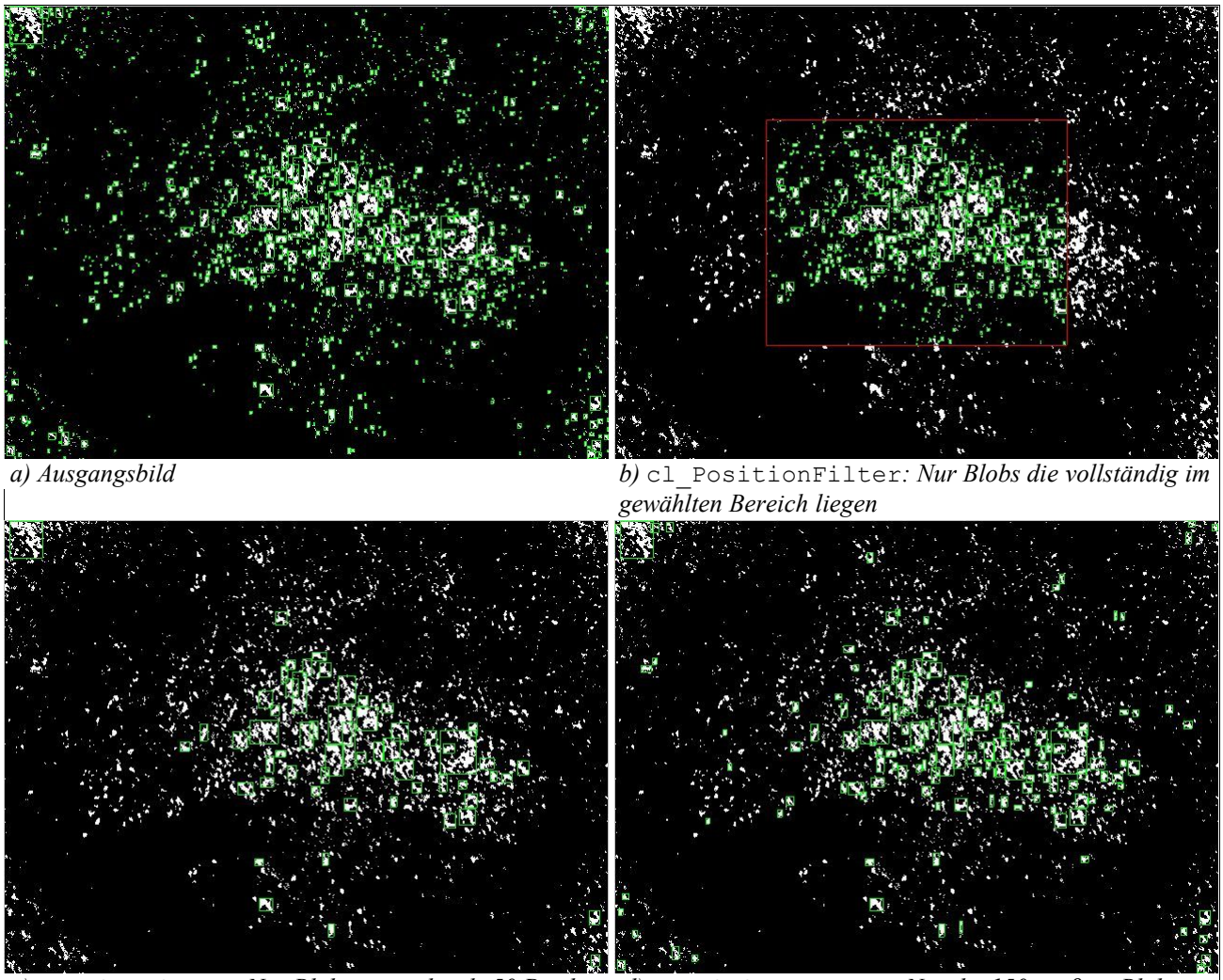
schnittlicher Pixelwert innerhalb des Bereich liegt, gelöscht werden oder verbleiben sollen.[1] Der Einsatz dieses Filters ist offensichtlich bei Blobs, die nicht aus Binärbildern gewonnen wurden, sinnvoll. Er kann vor allem dann benutzt werden, wenn über besonders gesuchte Blobs (beispielsweise solche, die zu einem bekannten Muster gehören) ungefähre Helligkeitswerte bekannt sind.

Es ist normalerweise sinnvoll, Blobs, die den Rand des aktuellen Bildes berühren, heraus zu filtern. Es besteht nämlich die große Wahrscheinlichkeit, dass die Blobs eigentlich nur zu einem Teil (verglichen mit Vorgänger- oder Nachfolgebildern) abgebildet sind. Damit wächst auch die Wahrscheinlichkeit einer Fehlidentifizierung der Blobs. Unter anderem zum Filtern solcher Blobs am Bildrand wurde ein anderer Blob-Filter implementiert.[2] Der Benutzer kann einen rechteckigen Bildausschnitt definieren und bestimmen, ob die gefilterten s – ganz oder teilweise – inner- oder außerhalb liegen sollen. Dieser Filter kann also auch benutzt werden, um s in anderen Bereichen auszublenden, z.B. wenn bekannt ist, dass die Kamera dort fehlerhafte Bilder liefert.
(siehe auch Abbildung III.7.b))

Speziell bei kleinen s ist die Wahrscheinlichkeit relativ hoch, dass andere s mit ähnlichen Eigenschaften existieren, und es daher bei der Identifizierung zu Fehlern kommt. Außerdem ist bei kleinen s die Wahrscheinlichkeit höher, dass sie durch Rauschen oder andere Mess- bzw. Aufnahmefehler der Kamera entstanden oder stark gestört sind. Daher wurde ein weiterer Blob-Filter zur Verfügung gestellt, der Blobs nach ihrer Größe auswählt.[3] Auch bei diesem Filter kann der Benutzer den Bereich zulässiger Werte, in diesem Fall der Anzahl der im Blob enthaltenen Pixel festlegen. (siehe auch Abbildung III.7.c))

Ein Vorteil der drei bisher vorgestellten Filter ist es, dass die Bedingung, ob ein Blob gelöscht werden soll, unabhängig von anderen Blobs geprüft werden kann. Daher ist die Komplexität ihrer Algorithmen nur linear abhängig von der Anzahl der betrachteten Blobs. Diese Filter haben aber den Nachteil, dass nicht vorhersehbar ist, wie viele Blobs nach Anwendung der Algorithmen übrig bleiben. Sind dies zu wenige, so kann die Messung der Eigenbewegung letztendlich recht ungenau werden. Sind es aber weiterhin sehr viele, so ist die für die Identifizierung der Blobs benötigte Zeit immer noch entsprechend lang. Daher wurde ein weiterer Filter implementiert, der zwar auch kleine Blobs ent-

1 Der Filter ist implementiert in der Klasse `dip::blob::cl_MeanValueFilter`.
2 Dieser Filter ist implementiert in der Klasse `dip::blob::cl_PositionFilter`.
3 Die Klasse `dip::blob::cl_SizeFilter` ist die Implementierungdavon.



a) Ausgangsbild

b) `cl_PositionFilter`*: Nur Blobs die vollständig im gewählten Bereich liegen*

c) `cl_SizeFilter`*: Nur Blobs mit mehr als 50 Pixeln*

d) `cl_LimitNumBlobs`*: Nur die 150 größten Blobs*

Abbildung III.7: Anwendung verschiedener Blob-Filter auf ein Beispiel.

fernt, aber nach der Regel: Entferne alle, bis auf die x größten Blobs. Wobei x eine vom Benutzer fest zu legende Anzahl ist. Sind von vorn herein weniger Blobs vorhanden, so werden auch keine entfernt.[1] Da hier aber die Blobs zu einander in Beziehung gesetzt werden müssen, um herauszufinden, welches die „größten" Blobs sind, steigt auch die Komplexität des Filters. In der aktuellen Implementierung wird die aktuelle Liste der Blobs zunächst sortiert[2] und danach alle bis auf die x größten Elemente gelöscht. Die Komplexität entspricht also der des Sortieralgorithmus, welche $O(n)=n*\log(n)$ (mit n als Anzahl der Blobs) ist. (siehe auch Abbildung III.7.c))

1 Die entsprechende Filterklasse heißt `dip::blob::cl_LimitNumBlobs`.

2 Das Sortieren ist momentan implementiert mit Hilfe der, in der STL (Standard Template Libary) vorhandenen, Methode `sort()` für Listen. [9]

Und dann gibt es noch einen Blob-Filter, welcher identifizierte Blobs entfernt, deren gefundene Positionsänderung sich stark von der der jeweiligen Nachbarblobs unterscheidet.[1] Dies geschieht mit Hilfe desselben Algorithmus, der auch zur Filterung von Bildbewegungsdaten zur Anwendung kommt. Er wird in Abschnitt IV.2 ausführlich erläutert.

III.6. Blob-Identifizierung

Nach dem die Blobs aus dem Bild extrahiert und anschließend gefiltert wurden, werden sie identifiziert.[2] Hierzu merkt sich der Identifizierungsalgorithmus stets die Blobs aus dem letzten Aufruf und versucht, diese in der neuen Liste wieder zu finden. Um einen alten Blob in der Liste der neuen Blobs wieder zu finden, wird jedem in Frage kommenden Blob ein Wert zwischen Null und Eins zugeordnet, welcher die Ähnlichkeit der beiden Blobs repräsentiert. Es wird angenommen, dass der Blob mit der größten Ähnlichkeit der Gesuchte ist, falls er nicht einem anderen der alten Blobs noch ähnlicher ist. Neue Blobs, die nicht identifiziert werden können, werden aus der Liste der aktuellen gelöscht[3], alte Blobs, die nicht mehr wiedergefunden werden, merkt es sich nicht weiter.

Das Berechnen bzw. Zuordnen des Ähnlichkeitswertes übernehmen „Ähnlichkeitsschätzalgorithmen".[4] Jede vorliegende Implementierung von dieser Algorithmen schätzt die Ähnlichkeit unter einem anderen Gesichtspunkt. Deshalb ist es meist sinnvoll, mehrere unterschiedliche Schätz-Algorithmen einzusetzen. Jeder Schätz-Algorithmus gibt die Ähnlichkeit als einen Wert zwischen Null und Eins an. Diese Werte werden addiert. Damit der Gesamtähnlichkeitswert auch zwischen Null und Eins liegt, wird dieses Schätzergebnis vor der Addition noch mit einem individuellen Faktor – auch als „Relevanzwert" bezeichnet – multipliziert, wobei die Summe der Faktoren (deren Werte ebenfalls zwischen Null und Eins liegt) stets Eins ist. Diese Relevanzwerte können alle gleich sein[5], sie können aber auch zur Gewichtung dienen, so dass prägnantere oder stabilere

1 In der Implementierung ist das die Klasse `dip::blob::cl_LocalMovementFilter`.

2 Implementiert ist dieser Vorgang in der Klasse `dip::blob::cl_Identifyer`.

3 Wobei sie intern allerdings gespeichert werden, um sie bei der nächsten Identifizierung zur Verfügung zu haben.

4 Sie werden über die Schnittstelle `dip::blob::i_SimilarityEstimator` implementiert.

5 dann ist der Wert eines jeden Faktors $\frac{1}{n}$, mit n als Anzahl der verwendeten Schätz-Algorithmen.

Blobmerkmale bei der Ähnlichkeitsschätzung stärker berücksichtigt werden. Diese Zusammenhänge sind im nebenstehenden Kasten noch mal mathematisch zusammengefasst.

Seien B_1 und B_2 Blobs,
$S(B_1, B_2)$ ihr geschätzter Gesamtähnlichkeitswert,
$E_i(B_1, B_2)$ ihr vom i-ten Schätz-Algorithmus geschätzter Ähnlichkeitswert,
n die Anzahl der Schätz-Algorithmen
und r_i der i-te Relevanzwert.
Es gelte dann:

- $\forall\, j \in \mathbb{N} : E_j(B_1, B_2) \in [0;1] \;\wedge\; r_j \in [0;1]$
- $\sum_{i=1}^{n} r_i = 1$
- $S(B_1, B_2) = \sum_{i=1}^{n} r_i \cdot E_i(B_1, B_2) \in [0;1]$

Zur Verbesserung der Bearbeitungsgeschwindigkeit und der Schätzergebnisse können Schätz-Algorithmen Blobs, die sie für vollkommen „unähnlich“ bzw. „verschieden“ vom zu identifizierenden Blob halten, aus der Liste der potentiell in Frage kommenden Blobs löschen, sie quasi disqualifizieren. Dies geschieht dann unabhängig von bereits vergebenen Ähnlichkeitswerten. Hierzu kann der Benutzer normalerweise einen Grenzwert angeben, der besagt, ab wann ein Blob unter dem betrachteten Gesichtspunkt als nicht ähnlich gelten soll. (siehe auch Ablaufdiagramm auf Seite 30)

Ein implementierter Algorithmus schätzt die Ähnlichkeit anhand des Größenunterschiedes von zwei Blobs.[1] Der Benutzer kann hierfür angeben, um wie viele Pixel sich zwei Blobs unterscheiden dürfen, damit sie noch als ähnlich gelten. Blobs, die sich um mehr als diese Pixelanzahl vom zu findenden Blob unterscheiden, werden aus der Liste der potentiellen Blobs gelöscht. Der zugeordnete Ähnlichkeitswert für alle anderen Blobs berechnet sich so:

$$E(B_1, B_2) = 1 - \frac{A(B_1) - A(B_2)}{\Delta_{max}}$$, mit $A(B_i)$ als Pixelanzahl vom i-ten Blob und Δ_{max} als der benutzerdefinierten maximalen Pixeldifferenz.

Blobs, die genau gleich viele Pixel beinhalten, bekommen also den Ähnlichkeitswert Eins zugeordnet. Entspricht die Differenz der Pixelanzahl zweier Blobs genau der benutzerdefinierten Maximalen, so wird als Ähnlichkeitswert Null ausgegeben.

1 Implementiert in der Klasse `dip::blob::cl_ BySizeEstimator`.

Das Schätzen der Ähnlichkeit von zwei Blobs anhand der absoluten Differenz ihrer Pixelanzahl hat allerdings den Nachteil, dass es die Ausgangsgröße des Blobs nicht berücksichtigt. Es ist schließlich viel wahrscheinlicher, dass es sich bei zwei Blobs, deren Pixelanzahl 100 und 105 ist, um dieselben, bzw. um sehr ähnliche Blobs handelt als bei zwei Blobs, die nur 3, bzw. 8 Pixel enthalten. Deshalb ist es besser, eine Ähnlichkeitsschätzung auf Basis der relativen Pixeldifferenz zweier Blobs durchzuführen. Diese Möglichkeit wurde auch zur Verfügung gestellt[1]. Der Benutzer kann also angeben, um welchen Faktor sich die Anzahl der enthaltenen Pixel zweier Blobs unterscheiden darf, damit sie als ähnlich gelten. In diesem Fall wir der zugeordnete Ähnlichkeitswert so berechnet:

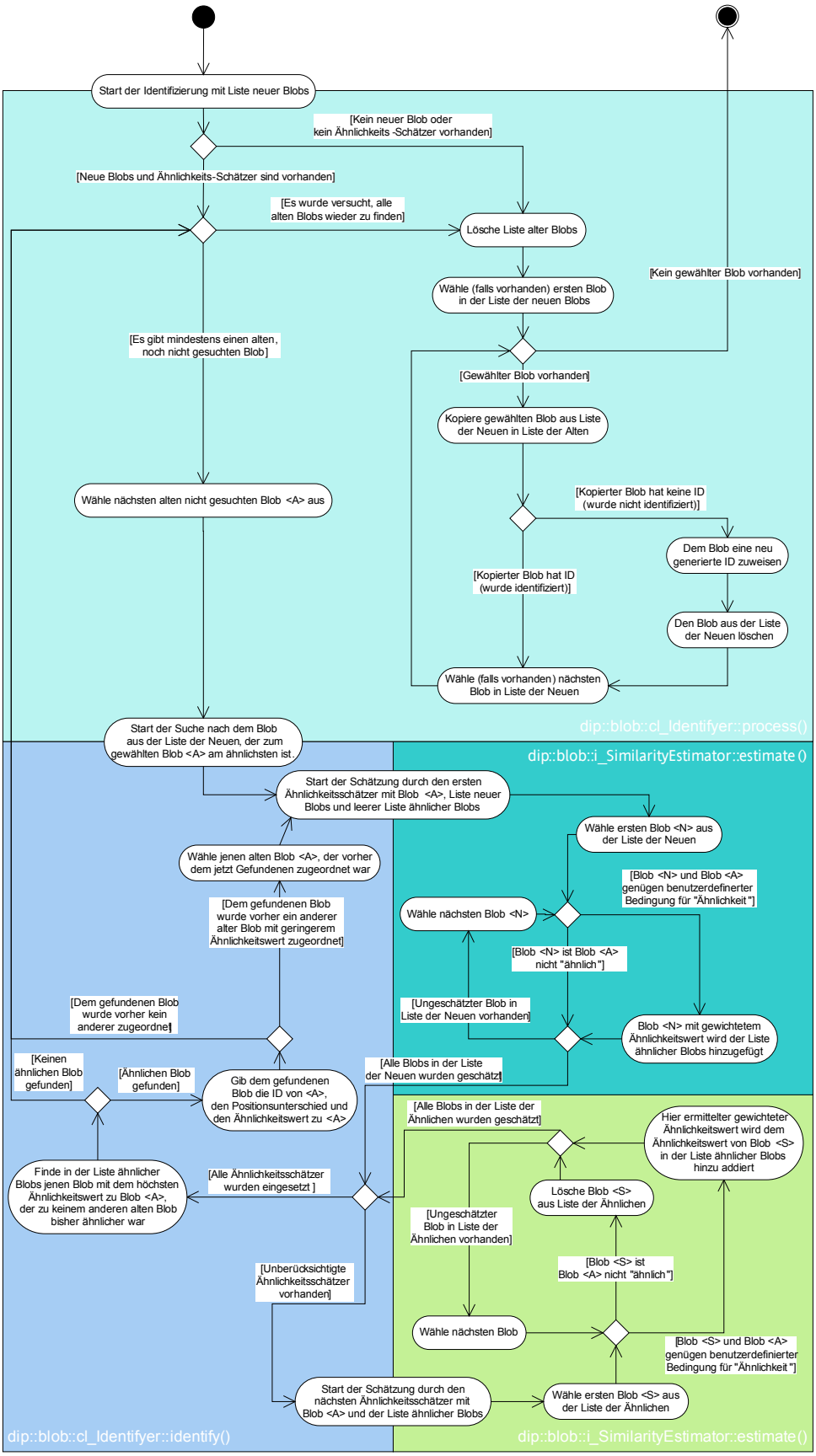
$E(B_1, B_2) = 1 - \frac{|A(B_1) - A(B_2)|}{A(B_1) \cdot \lambda_{max}}$, mit $A(B_i)$ als Pixelanzahl vom i-ten Blob und λ_{max} als benutzerdefiniertem Faktor.

Auch hier gilt für den ausgegebenen Wert: Ist die Pixelanzahl zweier Blobs gleich, so erhalten sie den Ähnlichkeitswert Eins. Zwei Blobs, deren Pixelanzahl sich genau um den benutzerdefinierten Faktor unterscheiden, erhalten den Ähnlichkeitswert Null.

Eine andere Schätzung wird aufgrund der Position der Blobs gemacht.[2] Wenn sich zwei Blobs sehr ähnlich sind, so ist es doch unwahrscheinlicher, dass es sich um dieselben handelt, je weiter sie auf den beiden Bildern, aus denen sie ermittelt wurden, von einander entfernt liegen. Dies gilt insbesondere, wenn die Bilder in einem sehr kurzen Zeitraum entstanden sind. Der Ähnlichkeitswert zweier Blobs wird in diesem Schätz-Algorithmus anhand der Positionen, die sie in den unterschiedlichen Bildern hatten, ermittelt. Der Benutzer kann die Entfernung angeben, die zwischen zwei Blobs maximal liegen darf, damit sie als ähnlich gelten.

1 Und zwar mit Hilfe der Klasse `dip::blob::cl_ByRelativeSizeEstimator`.

2 Realisiert in der Klasse `dip::blob::cl_ByPositionEstimator`.


Start der Identifizierung mit Liste neuer Blobs
[Kein neuer Blob oder kein Ähnlichkeits -Schätzer vorhanden]
[Neue Blobs und Ähnlichkeits-Schätzer sind vorhanden]
[Es wurde versucht, alle alten Blobs wieder zu finden]
Lösche Liste alter Blobs
Wähle (falls vorhanden) ersten Blob in der Liste der neuen Blobs
[Kein gewählter Blob vorhanden]
[Es gibt mindestens einen alten, noch nicht gesuchten Blob]
[Gewählter Blob vorhanden]
Kopiere gewählten Blob aus Liste der Neuen in Liste der Alten
Wähle nächsten alten nicht gesuchten Blob <A> aus
[Kopierter Blob hat keine ID (wurde nicht identifiziert)]
Dem Blob eine neu generierte ID zuweisen
[Kopierter Blob hat ID (wurde identifiziert)]
Den Blob aus der Liste der Neuen löschen
Wähle (falls vorhanden) nächsten Blob in Liste der Neuen
dip::blob::cl_Identifyer::process()
Start der Suche nach dem Blob aus der Liste der Neuen, der zum gewählten Blob <A> am ähnlichsten ist.
dip::blob::i_SimilarityEstimator::estimate()
Start der Schätzung durch den ersten Ähnlichkeitsschätzer mit Blob <A>, Liste neuer Blobs und leerer Liste ähnlicher Blobs
Wähle ersten Blob <N> aus der Liste der Neuen
Wähle jenen alten Blob <A>, der vorher dem jetzt Gefundenen zugeordnet war
[Blob <N> und Blob <A> genügen benutzerdefinerter Bedingung für "Ähnlichkeit"]
[Dem gefundenen Blob wurde vorher ein anderer alter Blob mit geringerem Ähnlichkeitswert zugeordnet]
Wähle nächsten Blob <N>
[Blob <N> ist Blob <A> nicht "ähnlich"]
[Ungeschätzter Blob in Liste der Neuen vorhanden]
[Dem gefundenen Blob wurde vorher kein anderer zugeordnet]
Blob <N> mit gewichtetem Ähnlichkeitswert wird der Liste ähnlicher Blobs hinzugefügt
[Keinen ähnlichen Blob gefunden]
[Ähnlichen Blob gefunden]
Gib dem gefundenen Blob die ID von <A>, den Positionsunterschied und den Ähnlichkeitswert zu <A>
[Alle Blobs in der Liste der Neuen wurden geschätzt]
[Alle Blobs in der Liste der Ähnlichen wurden geschätzt]
Hier ermittelter gewichteter Ähnlichkeitswert wird dem Ähnlichkeitswert von Blob <S> in der Liste ähnlicher Blobs hinzu addiert
Finde in der Liste ähnlicher Blobs jenen Blob mit dem höchsten Ähnlichkeitswert zu Blob <A>, der zu keinem anderen alten Blob bisher ähnlicher war
[Alle Ähnlichkeitsschätzer wurden eingesetzt]
Lösche Blob <S> aus Liste der Ähnlichen
[Ungeschätzter Blob in Liste der Ähnlichen vorhanden]
[Blob <S> ist Blob <A> nicht "ähnlich"]
[Unberücksichtigte Ähnlichkeitsschätzer vorhanden]
Wähle nächsten Blob
[Blob <S> und Blob <A> genügen benutzerdefinierter Bedingung für "Ähnlichkeit"]
Start der Schätzung durch den nächsten Ähnlichkeitsschätzer mit Blob <A> und der Liste ähnlicher Blobs
Wähle ersten Blob <S> aus der Liste der Ähnlichen
dip::blob::cl_Identifyer::identify()
dip::blob::i_SimilarityEstimator::estimate()

Eine weitere Eigenschaft eines Blobs ist seine Form. Diese wird sehr grob angenähert in einem weiteren implementierten Schätzungsverfahren betrachtet.[1] Es wird beurteilt, in wie weit sich das Verhältnis der Seiten des jeweils kleinsten umschließenden achsenparallelen Rechtecks (axis-aligned bounding-box) zweier Blobs unterscheidet. Also ob sie eine eher längliche oder quadratische Form haben. Auch hier darf der Benutzer die maximale Verhältnis-Differenz einstellen.

Die bisher vorgestellten Klassen stellen nur erste Implementierungen von Ähnlichkeits-Schätz-Algorithmen dar. Sie dienen zur Demonstration des Prinzips, nach dem Blobs identifiziert werden können. Sie alle haben aber den Nachteil, dass sie nach Gesichtspunkten urteilen, deren Änderung durch die eigentlich zu messende Eigenbewegung der Kamera vorprogrammiert ist. So verändert sich die Position eines Blobs wenn die Kamera parallel zum Boden verschoben, gedreht oder geneigt wird, bei Neigung, sowie Höhenänderung der Kamera ändert sich auch die Größe der Blobs und das Seitenverhältnis der Bounding-Box kann sich bei Neigung und Drehung ändern. Dem kann man teilweise begegnen, indem versucht wird, durch schnelle Bearbeitungszyklen die Änderungen pro Bild möglichst klein zu halten, aber auch dann sind Fehlidentifizierungen noch so häufig (was außerdem an der recht kleinen Zahl der betrachteten Identifizierungsmerkmale liegt), dass eine weitere Filterung der Daten notwendig wird. Zur Zeit geschieht das meist mit Hilfe von Filtern, welche nach dem in Abschnitt IV.2 beschriebenen Algorithmus vorgehen. Für die Zukunft ist es allerdings vorzuziehen, Filter zu erarbeiten, die Blobs nach weiteren Gesichtspunkten unterscheiden und identifizieren. Beispielsweise wäre ein weiteres Identifizierungskriterium das Verhältnis, in dem der Schwerpunkt eines Blobs seine Hauptachse teilt[2]. Hierzu muss natürlich zunächst eine Hauptachse für die Blobs ermittelt werden. Auch ist zu überlegen, ob nicht die ermittelte Bewegung aus dem Schritt davor nicht in die Blob-Identifizierung mit einzubeziehen ist, da es wahrscheinlich ist, dass sich auch die Bewegung von Schritt zu Schritt nur wenig ändert. Mehr Erläuterungen dazu und zu anderen Blob-Analyse-Algorithmen zum „relativen Hovern“ sind im folgenden Abschnitt III.7 „Weitere Blob-Analyse Algorithmen“ zu finden.

1 Realisiert in der Klasse `dip::blob::cl_BySideRelationEstimator`.

2 nur beeinflusst von der Neigungsänderung der Kamera zur Bodenfläche.

III.7. Weitere Blob-Analyse Algorithmen

Wie verschiedentlich schon erwähnt, stellen die implementierten Algorithmen zur Blob-Analyse nur einen kleinen Ausschnitt des Möglichen und Sinnvollen dar. Viele Algorithmen können erweitert, verbessert und ihre Implementierungen noch optimiert werden. Es wäre beispielsweise wünschenswert, das Auswahlkriteriums des Blob-Filters, der nur die „x größten Blobs“ passieren lässt, dahingehend verfeinern zu können, dass es „Gehört zu den x größten Blobs, die nicht den Bildrand berühren“ heißt. Ein äquivalentes Ergebnis lässt sich momentan erzielen, indem man diesem aktuell existierenden Filter, einen Positions-Blob-Filter voranstellt, dessen Auswahlbereich der Bildrand ist. Bei einem erweiterten Filter ist aber die Möglichkeit einer optimierten Implementierung gegeben, da dieser Blobs die schon zu klein sind, um zu den x Größten zu gehören, nicht mehr auf ihre Position zu überprüfen braucht. Auch der Algorithmus zum Suchen der x größten Blobs kann verbessert werden, z.B. indem man während des Sortierens schon die Blobs, die kleiner sind als die x „bisher“ Größten, von vorn herein löscht.

Eine andere Verbesserungsmöglichkeit ist auch die am Ende des vorherigen Abschnitts schon erwähnte Idee, die im vorherigen Schritt ermittelte Bewegung bei der Blob-Identifizierung mit einzubeziehen. Es ist wahrscheinlich, dass die Geschwindigkeit und Beschleunigung, die Bewegung also, sich in sehr kleinen Zeitabständen auch nur marginal ändert. Wie sich diese Bewegung auf einen Blob auswirkt, bzw. ausgewirkt hat, ist aber bekannt. So wird z.B. eine Bewegung der Kamera parallel zur Bildebene in einer Verschiebung der Blobs resultieren. Diese Verschiebung wird nun in sehr kleinen Zeitabständen – mit der Bewegung der Kamera – konstant bleiben. Man könnte daher diese Verschiebung zunächst auf die gemerkten Blobs des vorherigen Bildes anwenden, so dass eine Identifizierung der neuen Blobs einfacher fällt, weil sich die Blobs „ähnlicher“ geworden sind. Beim Berechnen der tatsächlichen Positionsänderung der Blobs muss diese Verschiebung dann natürlich rückgängig gemacht, bzw. berücksichtigt werden. Ebenso könnte man mit der Drehung, Skalierung und Verschiebung der Blobs, verursacht durch Kamera-Neigung, -Drehung oder -Translation senkrecht zur Bildebene, verfahren.

Auch gänzlich andere Blob-Such- und -Identifizierungsverfahren sind denkbar. Diese könnten zusätzlich direkt kombiniert werden: Eine Idee ist, sich zu Anfang nur eine bestimmte Anzahl Blobs zu merken, und diese stets im neuen Bild, an der (unter Einbeziehung der Bewegungswerte der Vergangenheit) ver-

muteten neuen Position wieder zu suchen, d.h. die eigentliche Blob-Suche, -Filterung und -Identifizierung zunächst auf einen kleinen Bildausschnitt für den einzelnen Blob zu begrenzen, und nur bei Bedarf auszuweiten. Nur wenn Blobs „aus dem Bild wandern“ oder aus anderen Gründen nicht wieder aufzufinden sind, wird für diese „Ersatz“ gesucht, vorzugsweise in Bildbereichen, in denen bis dahin wenige Blobs bekannt sind.

Genauso wie man sich zusätzliche Informationen über die Blobs (Hauptachse, Umriss, ...) merken kann, kann man aus ihnen weitere Informationen über die Bewegung der einzelnen Blobs gewinnen. Mit Hilfe der Hauptachse könnten z.B. Drehungen der Blobs gemessen werden, aus der Pixelanzahl kann man unter Umständen (vor allem im Zusammenspiel mit Umrissdaten) Skalierungsvorgänge erkennen. Diese zusätzlichen Informationen über Zustandsänderungen der Blobs könnten in eine erweiterte, verbesserte Eigenbewegungsschätzung einfliessen, oder auch bei bekannten Abhängigkeiten der Blobs (liegen alle in einer Ebene, verändern ihre Lage zueinander nicht, usw.) zur Verbesserung und Validierung der Blob-Identifizierung dienen.

IV. Eigenbewegung

IV.1. Bewegungen speichern und auswerten

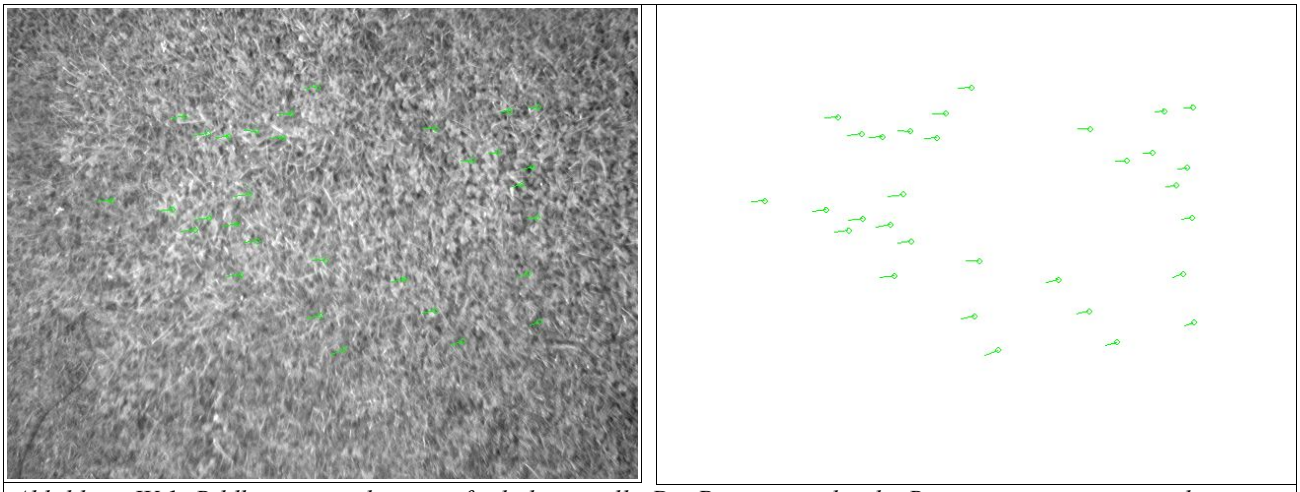


Abbildung IV.1: Bildbewegungsdaten grafisch dargestellt. Die Position zu der die Bewegung gemessen wurde ist durch einen Kreis markiert, die Bewegung als Verbindungslinie zur alten Position eingezeichnet.

Nachdem in der Blob-Analyse oder mit Hilfe anderer Algorithmen Bewegungen „im Bild“ festgestellt wurden, ist es möglich hieraus die Eigenbewegung der Kamera bzw. des Helicopters zu bestimmen.

Damit die gemessenen Bewegungen unterschiedlicher „Dinge“, Objekte[1] in den Bildern, zum Ermitteln der Eigenbewegung genutzt werden können, müssen die Bewegungsdaten in einem einheitlichen Format gespeichert werden. Primär ist es notwendig die zuletzt gemessene Bewegung des Objektes in Pixelkoordinaten und dessen aktuelle Position im Bild zu speichern. Sind $\vec{p}_i$ und $\vec{m}_i$ die Position im bzw. die Bewegung zum i-ten Bild, so gilt also $\overrightarrow{p_{i-1}}+\vec{m}_i=\vec{p}_i$ bzw. $\vec{p}_i-\vec{m}_i=\overrightarrow{p_{i-1}}$. Im implementierten Datenformat[2] können die messenden Algorithmen außerdem mit Hilfe einer numerischen Variable (zwischen 0 und 1) zu jedem Bildbewegungsdaten-Objekt angeben, wie sicher sie sich sind, dass die gemachte Messung richtig war. Der vorgestellte Blob-Analyse-Algorithmus setzt diesen Wert der „Messsicherheit“ beispielsweise auf den ermittelten Ähnlichkeitswert, denn je ähnlicher ein neuer und ein alter Blob erscheinen, desto sicherer ist sich die Blob-Analyse, dass der neue Blob richtig identifiziert wurde, und damit, dass die gemessene Bewegung korrekt ist.

1 z.Zt. Punkte oder Blobs, aber auch Kanten, spezifische Muster usw. denkbar.
2 `dip::mdp::cl_MeasuredMovement`

Die Interpretation der Bildbewegungsdaten geschieht in zwei Stufen[1]. Zuerst werden, falls notwendig, die Bildbewegungsdaten gefiltert, nicht so sehr um deren Zahl zu vermindern, sondern um Ausreißer und Fehlmessungen zu beseitigen. Diese können aus unterschiedlichen Gründen entstanden sein. Zum einen weil die Filter, die diese Daten erstellt haben, tatsächlich Punkte, Objekte oder Bildbereiche falsch zu geordnet haben. Zum anderen können Störungen und Rauschen bei der Aufnahme und Weitergabe der Bilder durch die Hardware dafür verantwortlich sein. Außerdem können auch tatsächlich „falsche" Bewegungen in den Bildern vorhanden sein, z.B. die Antenne oder der Schatten des Hubschraubers. Dinge also, die nicht das eigentlich Interessante, den Boden oder das Objekt zu dem relativ „gehovert", bzw. die Messung der Eigenbewegung vorgenommen werden soll, darstellen. Die Filterung der Bildbewegungsdaten übernehmen Bildbewegungsfilter.

Nach der Filterung findet die eigentliche Schätzung der Eigenbewegung relativ zum von der Kamera Aufgenommenem statt.

Sowohl die verwendeten Bildbewegungsfilter-Algorithmen, als auch der verwendete Eigenbewegungs-Schätz-Algorithmus können vom Benutzer bestimmt werden. Die Bildbewegungsfilter lassen sich dabei, wie auch schon die Blob-Filter zur Laufzeit in Reihenfolge, Art und Zahl verändern, und parametrisieren. Ebenso kann der Schätz-Algorithmus zur Laufzeit ausgetauscht und parametrisiert werden.

Alle derzeit implementierten Algorithmen zur Schätzung der Eigenbewegung gehen davon aus, dass das Objekt bzw. die betrachtete Fläche zu der die relative Eigenbewegung festgestellt werden soll, eben ist, und dass die übergebenen Bildbewegungsdaten zu Punkten bzw. Objekten die in dieser Ebene liegen gehören. Die Algorithmen müssen mit der relativen Ausgangsposition und -lage zu der Ebene initialisiert werden (insbesondere mit dem Abstand und den Neigungswinkeln).

In den folgenden Abschnitten (IV.2 und IV.3)werden nun die ersten implementierten Bildbewegungsfilter- und Eigenbewegungs-Schätz-Algorithmen erläutert. Und in Abschnitt IV.4 werden Konzepte und Ideen vorgestellt, nach denen in Zukunft weitere Algorithmen implementiert werden könnten.

1 Realisiert in der Klasse `dip::mdp::cl_PositionEstimator`.

IV.2. Bewegungen filtern

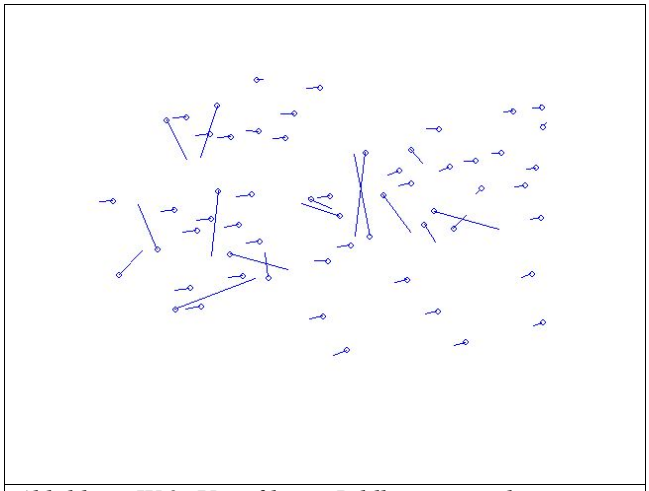


Abbildung IV.2: Ungefilterte Bildbewegungsdaten mit Ausreißern.

Viele der Algorithmen zur Schätzung der Eigenbewegung liefern besonders schlechte Ergebnisse, wenn es unter den Bildbewegungsdaten Ausreißer gibt. Wie schon im vorherigen Abschnitt erwähnt, wurden die Bildbewegungsfilter daher erstellt, um solche Ausreißer und Messfehler zu finden und zu eliminieren.

Das entscheidende Charakteristikum von Ausreißern ist, dass sie anders sind, dass sie sich stark von dem aufgrund der meisten anderen gemessenen Ergebnisse erwarteten Resultat unterscheiden. Der Unterschied zweier Bewegungen im i-ten Bild $\vec{m}_{i_a}, \vec{m}_{i_b}$ lässt sich ausdrücken als Betrag ihrer Differenz: $\left|\vec{m}_{i_a} - \vec{m}_{i_b}\right|$.

Falsch wäre nun die triviale Annahme, dass bei größeren Unterschieden notwendigerweise eines der beiden Beteiligten Messergebnisse falsch bzw. ein Ausreißer ist. Denn es können in den Bildern durchaus gegensätzliche Bewegungen, z.B. erzeugt durch Drehung der Kamera um die Bildmitte, oder auch unterschiedliche Längen der Bewegungsvektoren, z.B.

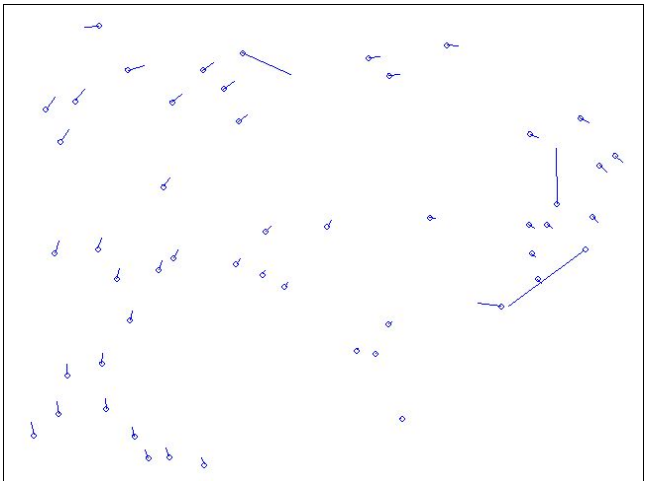


Abbildung IV.3: Ungefilterte Bildbewegungsdaten einer Drehung der Kamera mit Ausreißern.

hervorgerufen durch Neigung der Kamera relativ zum Untergrund, vorkommen und für die Schätzung der relativen Eigenbewegung wichtig sein.

Geht man aber davon aus, dass die Messung bzw. Schätzung der Eigenbewegung anhand einer betrachteten starren, relativ ebenen Fläche erfolgt, so ist der mögliche Unterschied zwischen den Bildbewegungen in zwei Punkten jeweils abhängig vom Abstand der Punkte zueinander. Der Unterschied von zwei lokal benachbart gemessenen Bewegungen ist immer sehr gering. Im Bild weiter voneinander entfernt gemessene Bewegungen können sich stärker unterscheiden. Ausreißer und ungewollte Messergebnisse haben also die Eigenschaft, dass sie sich stärker von den lokal benachbart gefundenen Ergebnissen unterscheiden, als reguläre, „richtige" Messungen. Es wird daher das arithmetische Mittel der gewichteten Unterschiede einer gemessenen Bewegung zu allen anderen Bewegungen betrachtet, wobei die Gewichtung anhand des Abstandes der Messpunkte im Bild erfolgt. Ergänzt wird die Gewichtung noch um den Wert der Messsicherheit angibt. Damit werden die Abweichungen zu „unsicheren" Messergebnissen nicht so stark berücksichtigt, wie die zu „sicheren".

Sind $\overrightarrow{p_{i_a}}$ und $\overrightarrow{p_{i_b}}$ die Ortsvektoren zu den Punkten in denen die Bewegungen gemessen wurden, so entspricht der Abstand dieser Punkte der Länge ihres Differenzvektors: $\left|\overrightarrow{p_{i_a}}-\overrightarrow{p_{i_b}}\right|$.

Das arithmetische Mittel der gewichteten Unterschiede zur k-ten gemessenen Bewegung, bei n_i insgesamt gemessenen Bewegungen im i-ten Bild, berechnet sich also wie folgt: $\frac{1}{\sum_{j=0}^{n_i} c_{i_j}} \cdot \sum_{j=0}^{n_i} \frac{\left|\overrightarrow{m_{i_k}}-\overrightarrow{m_{i_j}}\right| \cdot c_{i_j}}{\left|\overrightarrow{p_{i_k}}-\overrightarrow{p_{i_j}}\right|}$ [1], mit c_{i_j} als Messsicherheit der j-ten Messung im i-ten Bild.

Bei Ausreißern ist dieser Wert überdurchschnittlich hoch. Daher werden alle Bewegungsdaten für die dieser Wert überdurchschnittlich hoch ist gelöscht, so dass damit die Ausreißer entfernt werden. Allerdings werden dabei meist zu

1 Auf den Vorfaktor $\frac{1}{\sum_{j=0}^{n_i} c_{i_j}}$ wird in der konkreten Implementierung verzichtet, da er natürlich für alle Daten eines Bildes gleich ist, am Ergebnis eines Vergleichs der Werte also nichts ändert.

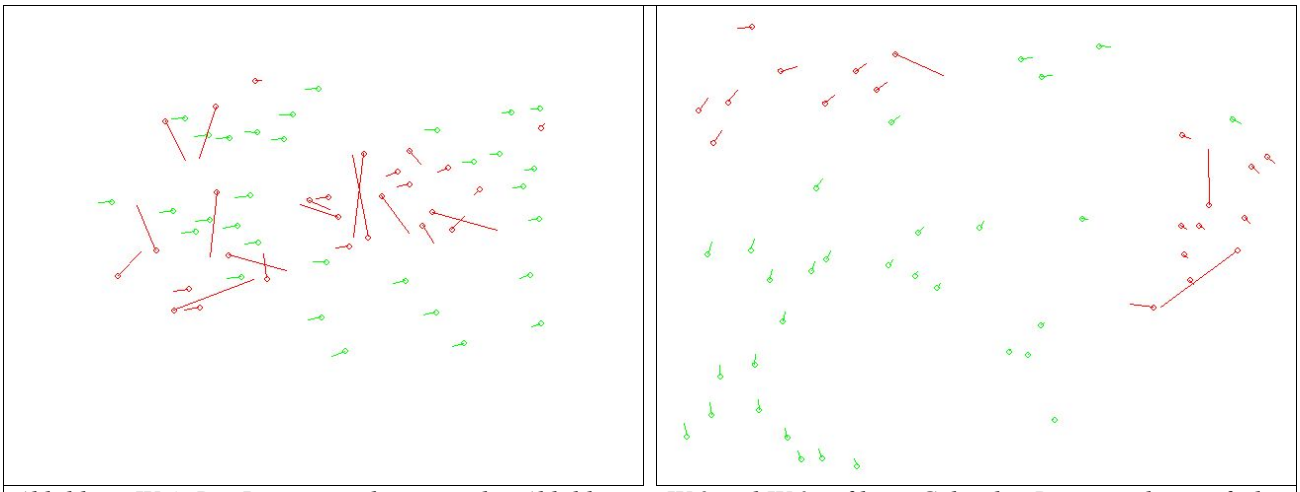


Abbildung IV.4: Die Bewegungsdaten aus den Abbildungen IV.2 und IV.3 gefiltert. Gelöschte Daten sind rot gefärbt.

viele Daten gelöscht, da auch der entsprechende Wert von Bewegungsdaten, die sehr nahe an einem Ausreißer gemessen wurden überdurchschnittlich hoch sein kann. Dieses Verfahren sollte daher nur eingesetzt werden, wenn genügend Messdaten vorhanden sind.

Da große Ausreißer kleine hierbei „überdecken" können, wird dieses Verfahren in zwei Implementierungen[1] iterativ mehrfach hintereinander angewandt. Der Benutzer kann hierzu mehrere Abbruchbedingungen für die Iteration bestimmen, wie z.B. dass eine gewisse Mindestanzahl an Bewegungsdaten unterschritten wird oder dass der Durchschnittswert des arithmetischen Mittel der gewichteten Unterschiede der Bewegungsdaten zueinander sich um weniger als einen Grenzwert verändert hat, dass also der letzte Iterations-Schritt nicht mehr genügend „Effekt" erzielt hat.

In einer weiteren Implementierung[2] werden alle bis auf die x Bewegungsdaten mit dem niedrigsten mittleren gewichteten Unterschied gelöscht. Das ist zwar etwas schneller, bringt in der Praxis aber nicht dieselben guten Ergebnisse wie der iterative Ansatz.

1 Die Klassen dip::mdp::cl_SmoothLocalMoves und dip::blob::cl_LocalMovementFilter implementieren den Algorithmus iterativ. Letztere ist dazu gedacht identifizierte Blobs zu filtern, bevor deren gemessene Bewegung weitergegeben wird.

2 dip::mdp::cl_LimitNumMoves

IV.3. Erster Algorithmus zur Schätzung der Eigenbewegung

Nachdem die Algorithmen zur Blob-Analyse und Bildbewegungsfilterung erdacht und vorläufig fertiggestellt waren, wurde, zum Testen dieser, ein erster Algorithmus zur Schätzung der relativen Eigenbewegung implementiert.[1] Bei der Auswahl des Verfahrens wurde darauf geachtet, dass sich auch Zwischenergebnisse gut auf das jeweilige Bild beziehen und visualisieren lassen, damit leichter sichtbar wird, welche Parametrisierung und die Auswahl welcher Algorithmen inwieweit Einfluss auf die Schätzung der Eigenbewegung haben.

Es werden die für das relative Hovern besonders wichtigen Bewegungen, die Translationen in alle drei Richtungen sowie die Rotation um eine Senkrechte zur Bildebene betrachtet. Dabei wird davon ausgegangen, dass Punkte, zu deren Abbildern in der Bildebene Bewegungen vorliegen, in der Realität in einer Ebene liegen und ihre Lage relativ zueinander nicht verändern. Ausserdem wird von einem „Lochkamera"-Modell ausgegangen, d.h. es wird vorausgesetzt, dass durch die Optik keine Verzerrungen bei der Abbildung der Welt auf die Bildebene auftreten. Sind solche Verzerrungen vorhanden, so wären sie vorher rechnerisch auszugleichen. Letzte Voraussetzung ist noch, dass der anfängliche Abstand von der Kamera zur betrachteten Ebene bekannt ist.

Zunächst wird die Translation parallel zur Bildebene ermittelt. Diese hat meist den größten Anteil an den gemessenen Bewegungsvektoren, würde die Messung der anderen Bewegungsanteile also erheblich stören, wird aber von diesen kaum bzw. selten gestört. Werden Bild- und betrachtete Ebene parallel gegeneinander verschoben, so bewegen sich alle Bildpunkte gleichermaßen in dieselbe Richtung. Die beiden übrigen Bewegungen (Translation senkrecht zur Bildebene, Rotation um eine Senkrechte zur Bildebene) erzeugen punktsymmetrische Bildbewegungen. Daher wird der Mittelwert aller gemessenen Bildbewegungsvektoren als der von der Translation parallel zur Bildebene verursachte Anteil der Bildbewegung im i-ten Bild ($\overrightarrow{m_{xyi}}$) angenommen. Sind die betrachteten Bildpunkte hinreichend regelmäßig verteilt, so fällt damit jeglicher Störeinfluss

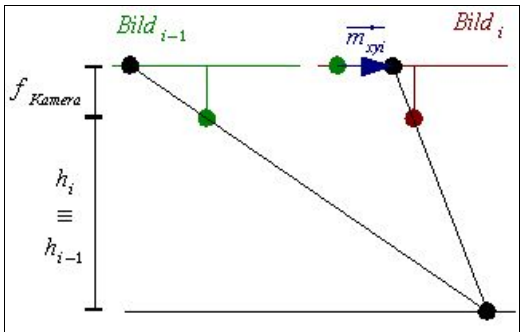


Abbildung IV.5: Translation parallel zur Bildebene

1 Die entsprechende Klasse heißt `dip::mdp::cl_SimpleMovesEstimator`.

durch die anderen Eigenbewegungen heraus. Bei der Mittelwertbildung werden Bewegungsdaten mit der angegebenen Messsicherheit gewichtet, so dass sicherere Daten stärker berücksichtigt werden. Es folgt also:

$$\overrightarrow{m_{xyi}} = \frac{\sum_{j=0}^{n_i} (\overrightarrow{m_{i_j}} \cdot c_{i_j})}{\sum_{j=0}^{n_i} c_{i_j}}$$

Mit Hilfe des Strahlensatzes und dem Abstand vom Boden im aktuellen (i-ten) Bild h_i lässt sich dann die Bewegung parallel zur Bildebene berechnen: $\frac{h_i \cdot \Delta_{Pixel}}{f_{Kamera}} \cdot \overrightarrow{m_{xyi}}$, mit f_{Kamera} als Brennweite der Kamera und Δ_{Pixel} als Pixel-Abstand auf der Bildebene.

Für die weiteren Berechnungen wird $\overrightarrow{m_{xyi}}$ von allen gemessenen Bildbewegungen subtrahiert, so dass nur die von der Translation senkrecht zur Bildebene und der Rotation um eine Senkrechte zur Bildebene verursachten Anteile in den Bildbewegungsvektoren übrig bleiben. Diese stehen senkrecht aufeinander und sind daher voneinander unabhängig.

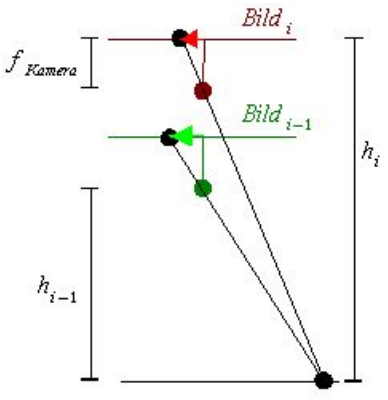


Abbildung IV.6: Translation senkrecht zur Bildebene

Die Translation senkrecht zur Bildebene drückt sich in Bildbewegungen („sternförmig“) hin zu bzw. weg von der Bildmitte aus. Ebenfalls laut Strahlensatz ist dabei das Verhältnis, in dem alter und neuer Abstand des betrachteten Bildpunktes von der Bildmitte stehen, gleich dem Verhältnis von neuer zu alter Höhe. Um Ungenauigkeiten und Fehlmessungen auszugleichen, wird dieses Verhältnis von altem zu neuem Abstand zur Bildmitte von allen Messungen wiederum jeweils mit der Messsicherheit gewichtet und gemittelt.

Nach Abzug der Translationsanteile von den gemessen Bildbewegungen wird noch die Rotation um eine Senkrechte zur Bildebene ermittelt. Diese Rotation hat normalerweise den kleinsten Anteil an den erzeugten Bildbewegungen. Sie ist daher in diesem Verfahren besonders fehleranfällig und sensibel gegen Ausreißer. Normalerweise sollte die verbleibende Rotation eine Rotation um die Bildmitte sein. Aufgrund von Mess- oder Rechenfehlern (z.B. weil die gemessenen Bewegungen bei Schätzung der Translation parallel zur Bildebene nicht

gleich verteilt waren) kann der Mittelpunkt der Rotation aber auch verschoben sein. Es handelt sich dann also um eine verbliebene Kombination aus Translation parallel zur Bildebene (entsprechend dem Differenzvektor zwischen Bildmittelpunkt und Rotationsmittelpunkt) und Rotation um die Bildmitte.

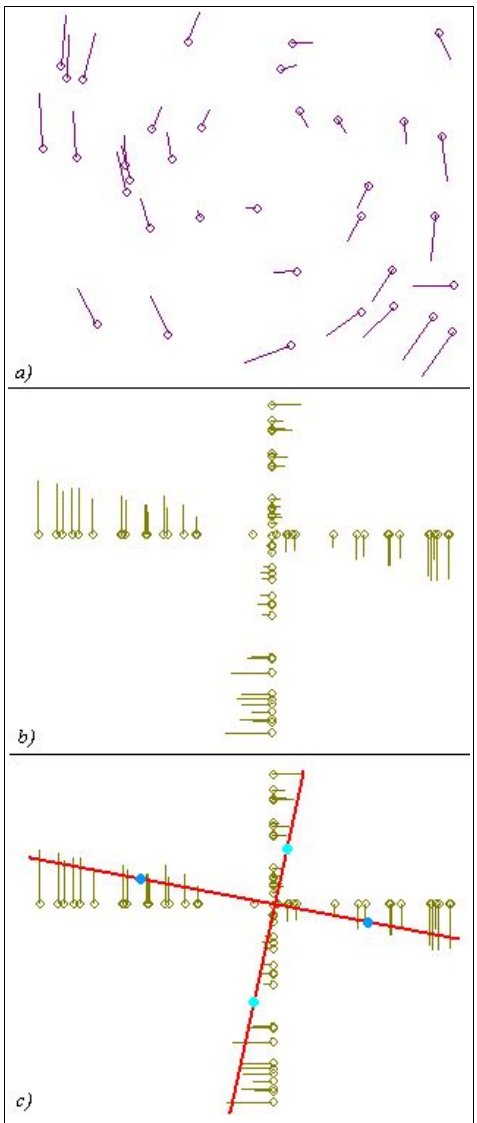


Abbildung IV.7 Bestimmung von Rotationswinkel und -mittelpunkt

Es ist nun der Mittelpunkt einer solchen Rotation und der Rotationswinkel zu bestimmen. Die verbliebenen Bildbewegungsvektoren sollten jeweils die Eigenschaft haben, senkrecht zu der Verbindungslinie zwischen ihrer Messposition und dem Rotationsmittelpunkt zu stehen (Abbildung IV.7.a). Sie werden nun zerlegt, und zwar wird der x-Anteil des Bewegungsvektors an der y-Position, an der er gemessen wurde, der y-Anteil des Bewegungsvektors an der x-Position, an der er gemessen wurde, angelegt (Abbildung IV.7.b). Wert und Position der x-Anteile rechts und links der y-Achse werden jeweils gemittelt, ebenso Wert und Position ober- bzw. unterhalb der x-Achse. Durch die beiden gemittelten Werte wird jeweils eine Linie gelegt (Abbildung IV.7.c), wo die Linie der x-Anteile die y-Achse schneidet, ist der y-Wert des Mittelpunktes der Rotation abzulesen, und entsprechend der x-Wert des Mittelpunktes, wo die Linie der y-Anteile die x-Achse schneidet. Der Rotationswinkel lässt sich mit Hilfe des Arcustangens aus den Steigungen dieser Linien (bzw. des Kehrwertes) ermitteln.

IV.4. Weitere Algorithmen zur Schätzung der Eigenbewegung

Der im vorherigen Abschnitt beschriebene, erste implementierte Algorithmus zur Schätzung der Eigenbewegung hat den Vorteil, dass seine (Zwischen-) Ergebnisse besonders gut im Bild visualisiert werden können, da der Algorithmus die Bildbewegung entsprechend der geschätzten ursächlichen Eigenbe-

wegungen zerlegt und dabei bis zu einem sehr späten Zeitpunkt innerhalb der Ausführung in Bildkoordinaten rechnet, so dass keine Rücktransformation und Projektion der Ergebnisse in die Bildebene zur Visualisierung notwendig sind. Zudem ist der Algorithmus recht schnell, seine Komplexität ist nur linear abhängig von der Anzahl der Eingangsbildbewegungen. Aber der Algorithmus ist relativ ungenau, durch die einfache Mittelwertbildung zur Verwertung aller übergebenen Daten anfällig gegenüber Ausreißern[1] und berücksichtigt nur vier der sechs Freiheitsgrade im Raum, die Neigung der Bild- und der betrachteten Ebene gegeneinander bleibt unbeachtet.

Daher werden andere bessere Algorithmen zur Schätzung der Eigenbewegung benötigt. Ein weiterer wird zur Zeit schon implementiert. Der geht auch von einer betrachteten Ebene aus und optimiert zur Schätzung der Eigenbewegung ein überbestimmtes lineares Gleichungssystem, so dass der Fehler minimal wird. Er ist in der Lage mit Hilfe von mindestens acht gemessenen Bildbewegungen die Eigenbewegung in allen sechs Freiheitsgraden im Raum zu schätzen. Weitere, auch schon erforschte, Schätzalgorithmen, unter anderem auf Basis eines Kalman-Filter, sollen folgen. [2][10][21]

Es sind auch Schätzalgorithmen zur Bestimmung der Eigenbewegung über unebenem Untergrund geplant, hierzu ist aber eine Erweiterung der gespeicherten Daten zu einer Bildbewegung notwendig, es muss nämlich zu jedem Punkt auch sein aktueller Abstand zu Kamera, bzw. seine Position in der Welt gespeichert werden. Dazu müssen die Positionen der Punkte in der Welt des Anfangs bzw. eines Initialisierungsbildes oder -musters bekannt sein und die Positionen der neu gefundenen Punkte in der Welt stets anhand dieser ermittelt werden.

Eine andere Idee für ein Verfahren ist, die Bildbewegung aus der Überlagerung mehrerer zeitlich aufeinander folgender Bilder zu schliessen. Besonders helle oder dunkle Punkte in diesen Bildern sollten dabei Linien, Spuren, Kanten hinterlassen, anhand deren Form und Ausrichtung sich ebenso die relative Eigenbewegung ablesen lassen sollte.

1 Dies gilt insbesondere, wenn die Ausreißer einen durchschnittlichen oder hohen Messsicherheitswert haben, z.B. weil der Algorithmus, der die Daten ermittelte, keine individuelle Messsicherheit ausgibt und den Messsicherheitswert auf einen Standardwert gesetzt hat.

V. Das dip-Framework

Die in den vorangegangenen Kapiteln beschriebenen Algorithmen wurden im Rahmen des dip-Frameworks (digital image processing) implementiert. Das dip-Framework ist eine Softwareumgebung für die digitale Bildverarbeitung im ARTIS Projekt.

Bei der Bildverarbeitung im ARTIS Projekt liegt der Schwerpunkt in der Erforschung neuer Algorithmen und dem Vergleich mit bestehenden Methoden. Es erschien daher sinnvoll, eine Softwareumgebung zu schaffen, in welcher diese Algorithmen einfach realisiert und unter kontrollierten Bedingungen auf ihre Geschwindigkeit, Zuverlässigkeit und Eignung getestet werden können. So entstand in Zusammenarbeit mit Olaf Guth, der an einer weiteren Diplomarbeit aus dem Themenbereich der digitalen Bildverarbeitung im ARTIS Projekt arbeitete, das dip-Framework. In diesem Kapitel werden das dip-Framework, sein Design und die softwaretechnischen Besonderheiten vorgestellt. [3]

V.1. Ein Überblick über das dip-Framework

Das dip-Framework ist ein objektorientiertes Framework auf Basis der Sprache C++. Es wurde eine objektorientierte Sprache gewählt, da diese die erforderlichen Aspekte der Vielseitigkeit, Flexibilität und Wiederverwendbarkeit elegant unterstützt und sich die grundlegenden Konzepte der Bildverarbeitung auch sehr deutlich abbilden lassen. Man kann die Funktionalität des Frameworks in einem Satz grob zusammenfassen:

> Es werden nacheinander Bilder von einem oder mehreren Filtern beliebiger Art sequentiell bearbeitet, wobei jeder Filter wiederum als Ergebnis mindestens ein Bild, je nach Typ aber auch zusätzlich andere Ausgaben liefert.

Im Framework spiegelt sich dieses z.B. in den grundlegenden Typen `i_Image` und `i_Filter` wider, welche die Bilder und Filter repräsentieren, oder auch in der Klasse `cl_FilterMaster`, in der die Filter erstellt und angeordnet werden können.

V.1.1. Motivation für ein neues Bildverarbeitungs-Framework

Auch wenn die digitale Bildverarbeitung noch eine ziemlich junge Disziplin in Forschung und Technik ist, gibt es aufgrund der vielfältigen Anwendungsmöglichkeiten und Einsatzgebiete bereits eine Vielzahl an kommerziellen und kostenlos verfügbaren Softwarebibliotheken, in denen die verschiedensten Filter und Verfahren zur Bildbearbeitung und -analyse fertig vorhanden sind. Man muss also das Rad nicht neu erfinden, sondern braucht diese nur in seine eigene Software einbauen und sich um die Details kaum noch kümmern. Dennoch wurde für das ARTIS Projekt ein eigenes neues Framework implementiert. Die Gründe dafür werden in diesem Abschnitt näher vorgestellt.

Der größte Nachteil vorhandener Bibliotheken ist für gewöhnlich ihr Umfang. Da die Programmierer dieser Bibliotheken die konkreten Rahmenbedingungen der Hardware und Aufgabenstellungen nicht kennen, müssen sie viel mehr Anwendungsfälle berücksichtigen und unterstützen, viel allgemeiner „denken", als es für den konkreten Fall im ARTIS Projekt nötig ist. Dies wirkt sich nicht nur auf den Umfang der (eingebundenen) Software, der Klassen und Funktionen aus, sondern auch auf die Geschwindigkeit. So können viele Algorithmen solcher Bibliotheken nicht nur Graustufen- sondern auch Farbbilder mit unterschiedlicher Farbtiefe bearbeiten, d.h. sie rechnen unter Umständen mit einer viel zu hohen Genauigkeit, müssen verschiedene Arten des Pixelzugriffs (ein, drei oder mehr Kanäle) beherrschen und jeweils entscheiden, welche davon angewandt werden muss. Dies alles kostet Zeit. Darüber hinaus kann es schwierig sein, die Verlässlichkeit und tatsächliche Genauigkeit von fremdimplementierten Algorithmen zu ermitteln, denn sie sind nur selten dokumentiert[1]. Es war daher besser, ein Framework zu entwickeln, das auf die aktuelle Kamera, Hardware und Aufgaben zugeschnitten ist. Gleichzeitig sollte es aber so modular und flexibel sein, dass es sich leicht an neue Aufgaben und Hardware anpassen lässt.

Einige der vorhandenen Bibliotheken haben auch den Nachteil, dass sie nur auf einer Plattform z.B. nur unter Win32 Betriebssystemen funktionieren. Zudem sind sie häufig auch auf die Analyse von Einzelbildern spezialisiert (z.B. zur Qualitätskontrolle oder Sortierung in der Fertigung oder Lagerhaltung großer Firmen).

1 aus einem Ergebnistyp „double" folgt schliesslich nicht automatisch, dass der Algorithmus mit 64-bit Genauigkeit rechnet

Viele große plattformunabhängige Bibliotheken wie OpenCV sind in funktionalen Programmiersprachen wie C geschrieben oder implementieren die Filter nicht als Objekte sondern als globale Funktionen. Diese müssen dann zusätzlich zu den zu bearbeiteten Bildern, bei jedem Aufruf erneut, die verschiedenen einstellbaren Parameter übergeben bekommen, was dazu führt, dass die Schnittstellen – die Signaturen dieser Funktionen – nicht einheitlich sind. Dadurch ist es nur mit großem Aufwand möglich, die Bearbeitungsreihenfolge, Zahl und Art der verwendeten Filter zur Laufzeit durch den Benutzer oder übergeordnete Programme zu ändern.[13]

Zwar gibt es auch schon einige objektorientierte Frameworks, wie z.B. LTI-Lib von der RWTH-Aachen. Diese sind aber, neben der oben angesprochenen Allgemeinheit, sehr konsequent objektorientiert (jeder Pixel ist ein Objekt), was mit großer Wahrscheinlichkeit zu weiteren Zeitverlusten führt, da häufig durchgeführte Speicherzugriffe wie z.B. die sequenzielle Abfrage der Pixelwerte länger dauern dürften. [14]

Leider haben frei verfügbare Bibliotheken häufig auch schlechte oder veraltete Dokumentationen. Da von der einzelnen Implementierung der Algorithmen stark abhängt, wie vertrauenswürdig deren Ergebnisse sind und mit welcher Effizienz (Geschwindigkeit, Genauigkeit, ...) diese erlangt werden, sollten fremdimplementierte Algorithmen erst nach eingehender Prüfung in sensible Kernbereiche einer Software übernommen werden, so dass die Zeitersparnis bei Verwendung dieser nicht so groß ist, wie es zunächst scheinen mag.

Dennoch ist es manchmal vorteilhaft, zur ersten Erprobung der generellen Eignung von bestimmten Algorithmen auf vorhandene Implementierungen zurückzugreifen. In einem selbst geschaffenen Framework sollten daher solche Fremdalgorithmen durch geeignete Kapselung einfach zu integrieren sein.

V.1.2. Die Anforderungen

Viele teils konträre Wünsche und Anforderungen liegen dem dip-Framework zugrunde. Da es sich um ein Experimentalsystem handelt, das auch in Zukunft zum Testen und Entwickeln von Bildverarbeitungsalgorithmen für teilweise noch nicht bekannte Aufgaben genutzt werden soll, ist Offenheit und Erweiterbarkeit sehr wichtig.

Mit Hilfe des Frameworks sollen zeitkritische Algorithmen – auch im realen Einsatz auf dem ARTIS Hubschrauber – getestet werden können. Es versteht sich daher fast von selbst, dass das Framework zum einen für administrative Aufgaben (z.B. Speicher- und Prozessverwaltung) so wenig Zeit wie möglich in Anspruch nehmen, zum anderen Typen, Schnittstellen und unterstützende Klassen für schnelle Algorithmen zur Verfügung stellen soll.

Aus der heterogenen Betriebsystemlandschaft (Linux auf dem Onboard-Rechner, Windows auf der Bodenstation und den meisten Entwicklungsrechnern) ergab sich die Forderung nach Plattformunabhängigkeit.

Die optimalen Parametereinstellungen für die erstellten Filter und Algorithmen – passend zur jeweiligen Aufgabe und Situation – müssen meistens noch gefunden werden. Daher ist es sinnvoll, wenn man auch während laufender Versuche die Parameter möglichst einfach manuell anpassen kann, so dass die Auswirkungen direkt an den Ergebnissen abgelesen werden können. Auch die Zahl, Art und Reihenfolge der verwendeten Filter sollten sich im laufenden Betrieb ändern lassen, z.B. um auf veränderte Rahmenbedingungen reagieren zu können, aber auch zum einfachen Experimentieren.

Es soll möglich sein, erfolgsversprechende Filterkonfigurationen ohne großen Aufwand auch in der Praxis testen zu können. Um Fehler und unterschiedliches Verhalten der Algorithmen zu vermeiden, ist es wünschenswert, dass bei Flugversuchen, Simulationen und Offlineanalyse stets dieselbe Implementierung der Algorithmen verwendet wird. Dabei sollen jeweils nur wirklich benötigte Frameworkmodule eingesetzt werden[1], um in jedem Fall die maximal mögliche Rechengeschwindigkeit zu erreichen.

Die digitale Bildverarbeitung ist ein multifunktional einsetzbares Werkzeug. Deswegen sollte sie auch parallel für unterschiedliche Zwecke verwendet werden können[2]. Dies kann bedeuten, dass mehrere Filter quasi parallel auf dasselbe Kamera- oder vorbearbeitete Bild zugreifen müssen. Dabei müssen Zugriffskonflikte und Inkonsistenz durch geeignete Mechanismen vermieden werden.

1 Z.B. ist eine Visualisierung auf dem Bordrechner während eines Flugversuchs nicht notwendig.

2 Z.B. könnte mit Hilfe der Bildverarbeitung ein gewisser Mindestabstand zum Boden gewahrt bleiben und während dessen nach dem Muster, welches den Landeplatz kennzeichnet, Ausschau gehalten werden.

OpenCV ist eine weit entwickelte[1] freie Bibliothek, die viele Filter und Algorithmen schon fertig implementiert mitbringt. Da es, wie schon im vorherigen Abschnitt erwähnt, vorteilhaft sein kann, zur ersten Erprobung der generellen Eignung von bestimmten Algorithmen auf schon implementierte Versionen zurück zu greifen, was unter anderem Olaf Guth für seine Diplomarbeit tun wollte, war eine weitere Anforderung an das Framework, OpenCV so gut wie möglich zu unterstützen, ohne dass die Implementierung[2] dabei abhängig von OpenCV sein sollte.[3][13]

Zusätzlich zu den Rahmenbedingungen für eine schnelle und einfache Implementierung von Filtern und Algorithmen soll das Framework Klassen und Werkzeuge bieten, welche die Experimente unterstützen. Diese messen z.B. die benötigte Rechenzeit, visualisieren Ergebnisse oder gewinnen zusätzliche, für die eigentlichen Algorithmen nicht benötigte Informationen[3], die zum Entwickeln und Debuggen von neuen Methoden hilfreich sind.

V.1.3. dip-Frameworkdesign: Konzepte und Richtlinien

Aus den im vorherigen Abschnitt beschriebenen Anforderungen resultierten bei Design und Implementierung des Frameworks verschiedene grundlegende Konzepte, Richtlinien und Festlegungen. Einige davon werden hier nun kurz vorgestellt.

Die aktuell im ARTIS Projekt verwendete Kamera erstellt monochrome 8-bit-Bilder. Infolge dessen wurde auch das dip-Framework auf 8-bit-Einkanalbilder ausgelegt[4] und optimiert, was den Pixel- und Bilddatenzugriff stark vereinfacht und beschleunigt.

Um Inkonsistenzen und Zugriffskonflikte bei parallel laufenden Algorithmen zu vermeiden wurde darauf geachtet, dass nur jeweils ein Objekt Bilddaten verändern kann. Dies ist in der Regel dasjenige Objekt, welches die Bilddaten erstellt hat. Andere Objekte können auf die Bilddaten nur lesend zugreifen.

---

1 wenn auch funktionale und schlecht dokumentierte

2 mit Ausnahme der entsprechenden Filter.

3 z.B. was durch einen Filter entfernt wurde

4 Dies ist übrigens nicht so unflexibel, wie es auf den Anschein haben mag: Mehrkanalbilder können z.B. in mehrere Einkanalbilder zerlegt werden, die dann mit den vorhandenen Algorithmen für Graustufenbilder weiterverarbeitet werden. Bei der digitalen Ver- und Bearbeitung von Farbbildern ist das gängige Praxis.

Da die Datenmenge eines Bildes in der Regel sehr groß ist, wurde das reine Kopieren dieser Daten, wo immer möglich, vermieden. Hierzu ist ein eigenes Speichermanagement für Bilddaten[1] implementiert worden. Bilddaten sollen deswegen nicht mehr verändert werden, nachdem sie anderen Objekten zugänglich gemacht wurden.

Abhängigkeiten von Klassen und Modulen werden minimal und nach Möglichkeit unidirektional gehalten. Dadurch können sehr flexibel und einfach dem jeweiligen Anwendungszweck angepasste Anwendungen erstellt werden, die entsprechend klein und schnell sind. So sind die Klassen, die Bilder repräsentieren, unabhängig von allen anderen Klassen, z.B. von den Filtern, und die Filterklassen sind unabhängig von den Kameraklassen, der GUI und untereinander.

Fast alle Klassen sind von wenigen grundlegenden Basisklassen abgeleitet. Das hat zur Folge, dass viele Funktionalitäten über einheitliche Schnittstellen angesprochen werden können, was die Flexibilität und (Wieder-) Verwendbarkeit der Implementierungen deutlich erhöht, da nur bei der Implementierung jener Methoden und Klassen auf die Spezifika eines bestimmten verwendeten Objektes eingegangen werden muss, die daran auch wirklich interessiert sind[2].

Zum Erreichen von Plattformunabhängigkeit wurden – soweit möglich – plattformunabhängige Befehle, Funktionen und Bibliotheken verwendet. Wenn auf plattformabhängige Funktionen und Bibliotheken zurückgegriffen werden musste, wurden diese entsprechend gekapselt und per Compilerschalter sichergestellt, dass für alle verwendeten Systeme compile- und lauffähige Versionen zur Verfügung stehen. Es wurde versucht, auch diese in ihrer Funktionalität übereinstimmen zu lassen[3].

Bei der Integration von OpenCV wurde ein Kompromiss eingegangen. Um eine schnelles Erzeugen von OpenCV spezifischen `IplImages` aus den Image-Objekten des Frameworks und umgekehrt zu ermöglichen, wurden entsprechende Methoden schon in den Image-Klassen des Frameworks zur Verfügung gestellt und diese Arbeit nicht den entsprechenden Filterklassen überlassen, was

1 ohne Geschwindigkeitseinbußen beim Datenzugriff (Siehe auch Abschnitt V.3.2)

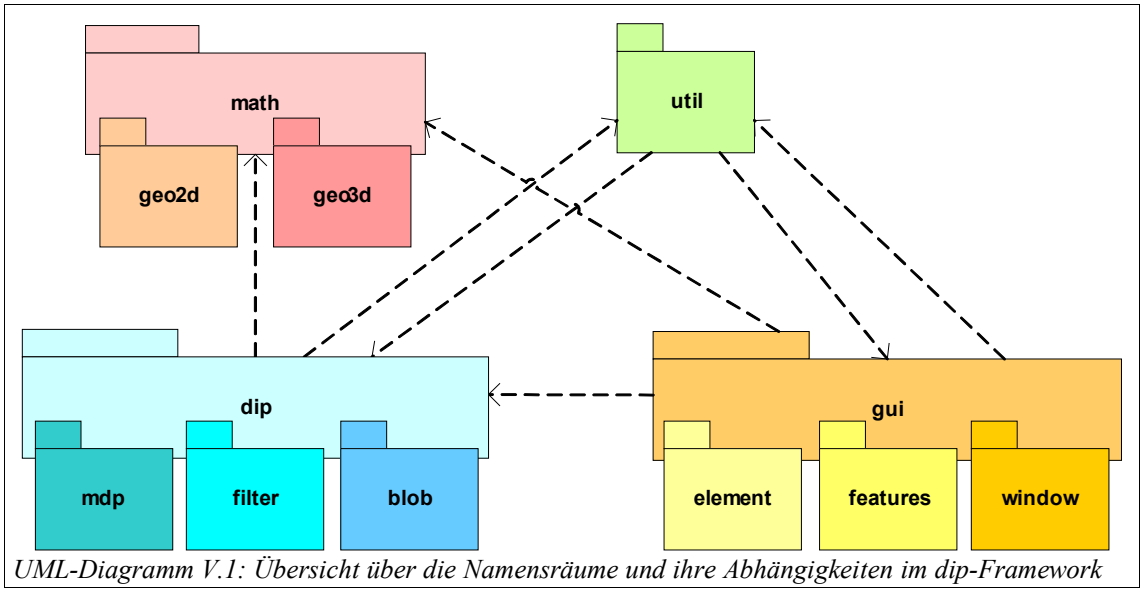
2 Es ist z.B. für die Implementierung einer Klasse `cl_FilterMaster` egal, welche Parameter die einzelnen Filter besitzen (siehe auch Abschnitt V.5 „Die Filterklassen und ihre dynamische Anordnung“).

3 Dies ist auch gelungen, nur nicht bei der „echten“ Kamera, hier fehlt noch ein passender Win32-Treiber.

eine leichte Verletzung der Forderung nach Unabhängigkeit des Frameworks von OpenCV und ähnlichen Bibliotheken darstellt. Da diese aber nur an einer zentralen Stelle auftritt und keine eigentlich OpenCV-unabhängigen Funktionen auf den eingefügten beruhen, ist die OpenCV-Unterstützung also auch wieder leicht zu entfernen und eine komplette OpenCV-Unabhängigkeit wiederherstellbar, falls die OpenCV-implementierten Filter eines Tages nicht mehr genutzt werden sollen. [13]

Ein eingeführter Mechanismus zur Gewinnung zusätzlicher – für die eigentlichen Algorithmen nicht benötigter – Informationen ist die so genannte „Debug Shell“. Dieses Konzept wurde schon für die „Bildbewegungs-Filter“ realisiert und wird in Abschnitt V.7 am Beispiel jener Implementierung näher erläutert. Es soll in Zukunft auch für andere Filterarten zum Einsatz kommen.

V.1.4. dip-Frameworkarchitektur: Die Namensräume



UML-Diagramm V.1: Übersicht über die Namensräume und ihre Abhängigkeiten im dip-Framework

Das Framework wurde nach Funktionalitäten in mehrere Module aufgeteilt, die jeweils ihren eigenen Namensraum besitzen. Das UML-Diagramm[1] bietet eine Übersicht über die vorhandenen Module und Untermodule. Es wurde zudem eingezeichnet, ob es Teile in einem Modul gibt, die von Teilen eines anderen Moduls abhängig sind[2].

1 Eine erklärende Übersicht zur verwendeten UML-Symbolik ist in Anhang VII.1 zu finden.

2 Es wurden hier, so wie in der gesamten Arbeit, nicht die Teile `math::statistics`, `dip::flow` und die Klasse `util::cl_Logger` berücksichtigt. Sie wurden nachträglich von Olaf Guth in das Framework eingefügt und spielen für die hier vorgestellten Verfahren und Algorithmen keine Rolle.[3]

Die Module selbst sind teilbar, d.h. es ist möglich, Anwendungen zu erstellen, die nur einzelne Teile beispielsweise von `util` enthalten[1].

Die vier Hauptmodule heißen `math`, `util`, `dip` und `gui`.

Das Modul `math` (mathematics) beinhaltet allgemein gültige Klassen und Konstanten für mathematische Berechnungen und Ausdrücke. In seinen Untermodulen `geo2d` (geometry of two dimensions) und `geo3d` (geometry of three dimensions) finden sich Klassen speziell für geometrische Berechnungen in der Ebene bzw. im Raum.

Im Modul `util` (utilities) sind alle Hilfs- und Administrationsklassen untergebracht. Einzelne Klassen und ihre Funktion aus `util` und `math` werden in Abschnitt V.2 „Die main()-Funktion und Hilfsmodule“ vorgestellt.

Das wohl wichtigste Modul ist `dip` (digital image processing). Es enthält alle Bildverarbeitungsklassen und -Algorithmen, angefangen von den Schnittstellen für Bildobjekte `i_Image` bis hin zu Algorithmen, die aus in Bilderfolgen gemessenen Bewegungen die Eigenbewegung der Kamera – und damit letztendlich die des ARTIS Helikopters – ableiten. Das Modul `filter` ist ein Teil von `dip` und enthält die Filterklassen, die auf die Bilder direkt angewendet werden. Es wird in Abschnitt V.5 „Die Filterklassen und ihre dynamische Anordnung“ näher beschrieben.

Weitere Teilmodule von `dip` sind `blob` und `mdp` (movement data processing). Auf die hier implementierten Algorithmen wurde in den Kapiteln III „Blob-Analyse“ und IV „Eigenbewegung“ eingegangen. Die Klassen, die Bilder repräsentieren, sind direkt im Modul `dip` zu finden. Mit ihnen und ihrer Implementierung beschäftigt sich Abschnitt V.3 „Die Bild-Klassen und -Schnittstellen“. Auch die Kameraklassen („reale“ und simulierte) sind Teil von `dip`. Sie werden zusammen mit dem Modul `gui` (graphical user interface) in Abschnitt V.8 „Kameraklassen und GUI“ kurz vorgestellt.

1 Die Teile können z.B. alle von `gui` unabhängig sein, so dass Anwendungen erstellt werden können, die ganz ohne das Modul `gui` auskommen.

V.1.5. Namens- und Farbkonventionen zu Implementierung und Codebeispielen

Beim Erstellen des Quellcodes wurden einige Namenskonventionen eingehalten.[1] Das Wissen darüber kann das Lesen und Nachvollziehen dieser Arbeit und der Codebeispiele erleichtern. In den Codebeispielen werden außerdem Codeelemente farblich unterschieden. Diese Konventionen werden hier zunächst erläutert und daran anschließend in einer Tabelle zusammengefasst.

Abstrakte Schnittstellen- und Oberklassennamen erhalten im Programmtext immer das Präfix „`i_`“ (wie interface), während alle sonstigen Klassennamen mit „`cl_`“ (class), Strukturnamen mit „`st_`“ (structure) und Aufzählungsnamen mit „`e_`“ (enumeration) beginnen. Namensräume, Aufzählungen, Strukturen, Klassen und Schnittstellen sowie Typen und C++-Schlüsselwörter werden in Codebeispielen `blau` dargestellt. All diese Namen und auch Funktionsnamen werden in „KamelNotation“ geschrieben. Funktionsnamen werden in Codebeispielen `rotbraun` eingefärbt. Variablen werden in „Unterstrich_Notation“ geschrieben. Konstanten und Aufzählungselemente werden ebenfalls in „Unterstrich_Notation“ geschrieben, aber nur in GROSSBUCHSTABEN. In den Codebeispielen sind Variablen und Konstanten `grau`, Aufzählungselemente und Präprozessormakros `violett` markiert. Konstanten erhalten das Präfix „`c_`“ (constant), Aufzählungselemente das Präfix „`E_`“ (enumeration element). Kommentare werden in Codebeispielen `grün` dargestellt, sie sind insbesondere in Header-Dateien so geschrieben, dass mit Hilfe des frei verfügbaren Tools „doxygen“[22] eine möglichst gute, leserliche und umfassende Dokumentation generiert werden kann.

Codeelement	Präfix	Notation	Anfangsbuchstabe	Farbe	Beispiel
Schnittstelle	`i_`	„KamelNotation“	groß	`blau`	`i_ImageUser`
Klasse	`cl_`	„KamelNotation“	groß	`blau`	`cl_MainWindow`
Struktur	`st_`	„KamelNotation“	groß	`blau`	`st_Point`

1 Jedenfalls sofern es sich um neu definierte Schnittstellen des dip-Frameworks handelt. Es gelten die beschriebenen Namenskonventionen natürlich nicht für Elemente verwendeter Fremdbibliotheken.

Codeelement	Präfix	Notation	Anfangsbuchstabe	Farbe	Beispiel
Aufzählung	`e_`	„KamelNotation“	groß	blau	`e_FilterType`
Namensraum	*kein*	„KamelNotation“	klein	blau	`math::geo2d`
Definierter Typ	*kein*	„Unterstrich_Notation“	klein	blau	`ip_entry`
Eingebauter Typ	-	-	-	blau	`double`
C++-Schlüsselwort	-	-	-	blau	`protected`
Funktion	-	„KamelNotation“	klein	rotbraun	`main()`
Methode	-	„KamelNotation“	klein	rotbraun	`pos->getX()`
Konstante	`c_`	„Unterstrich_Notation“, GROSSBUCHSTABEN	groß	grau	`c_RAD_TO_DEG`
Variable	*kein*	„Unterstrich_Notation“	klein	grau	`the_blob`
Aufzählungselement	`E_`	„Unterstrich_Notation“, GROSSBUCHSTABEN	groß	violett	`E_2THRESHOLDS`
Präprozessormakro	*kein*	„Unterstrich_Notation“, GROSSBUCHSTABEN	groß	violett	`BLOB_H`
Kommentare	-	-	-	grün	`// Debug Mode`

Die Farbkonventionen in Codebeispielen können bei Bedarf auch nochmals in Anhang VII.2 nachgeschlagen werden. Eine weitere tabellarische Übersicht der Präfixbedeutungen befindet sich in Anhang VII.3.

V.2. Die main()-Funktion und Hilfsmodule

Die Module `math` und `util` sind nicht für die digitale Bildverarbeitung oder deren Auswertung selbst verantwortlich, sie bieten vielmehr Klassen und Elemente, die entweder von den Bildverarbeitungsalgorithmen genutzt werden können, um beispielsweise mathematische Berechnungen durchzuführen, oder für die Erstellung lauffähiger Anwendungen notwendig sind. Eine `main()`-Funktion muss ebenfalls in jeder Anwendung vorhanden sein. Da diese Elemente für das Verständnis der Abläufe in den erstellten Algorithmen und Applikationen wichtig sind, werden sie im Folgenden etwas näher erläutert.

V.2.1. Der Namensraum für mathematische Konstrukte: math

Das Modul `math` ist, wie viele andere Teile des Frameworks auch, keinesfalls als fertig oder vollständig anzusehen. Es ist vielmehr ein „work in progress“, das genau jene mathematischen Typen und Konstanten beinhaltet, die gerade für die zu implementierenden Algorithmen von Nöten waren. So sind die Kreiskonstante π (`math::c_PI`) sowie Konstanten zur Umrechnung von Grad in Bogenmaß und umgekehrt vorhanden, die Eulersche Zahl aber nicht. Hilfreich sind auch die enthaltenen Klassen `math::geo2d::cl_2DVector` und `math::geo3d::cl_3DVector` welche Vektoren im Raum und in der Ebene repräsentieren. Sie wurden mit einer Vielzahl nützlicher Methoden und Operatoren ausgestattet, die z.B. die Multiplikation mit einem Skalar, Bildung des Kreuzproduktes zweier `cl_3DVector`-Objekte oder die Ausgabe der Länge eines Vektors ermöglichen. Auch hier kann in Zukunft noch Ergänzungsbedarf für weitere Operatoren und Methoden sein.

V.2.2. Anfang und Ende der Applikationsausführung: main()

In der Funktion `main()` startet und endet (hoffentlich) die Ausführung einer jeden Applikation. Um die Anwendungserstellung zu erleichtern, wurden Hilfsklassen – hauptsächlich im Modul `util` – erstellt, mit denen es möglich ist, die Funktion `main()` so schlank und übersichtlich wie möglich zu halten. Codebeispiel V.1 demonstriert dies an der `main()`-Funktion der Applikation SPICE.

```
#include "gui/fltk_updater/fltk_updater.h"
#include "gui/window/mainwindow.h"

int main()
{
   // init
   gui::window::cl_MainWindow::getMainWin();

   util::cl_UpdateManager* um=util::cl_UpdateManager::getMainUpdateManager();
   //main loop
   while (gui::cl_FltkUpdater::windowsDisplayed())
      um->update();

   //Singleton-Clean-Up:
   util::cl_SingletonManager::cleanUp();
   return 0;
}
```

Codebeispiel V.1: `main()`*-Funktion von SPICE*

Die `main()`-Funktionen von Applikationen, die mit Hilfe des dip-Frameworks realisiert sind, werden immer in etwa nach dem gleichen Schema ablaufen:

1. Zunächst wird das Hauptkontrollobjekt, welches die Aufsicht über alle ablaufenden Bildverarbeitungsvorgänge hat und mit der Außenwelt (anderen Programmen, Rechnern, Benutzern) kommuniziert, erstellt und gegebenenfalls durch an die Applikation übergebenen Aufrufparametern initialisiert.

2. Danach werden, solange eine Abbruchbedingung[1] nicht erfüllt ist, mit Hilfe des Update-Managers[2] zyklisch alle Objekte, die dessen bedürfen, aktualisiert, was in der Regel heißt, dass ein neues Bild von der Kamera empfangen und bearbeitet wird.

3. Zum Ende der Anwendung wird ggf. ein Abschlussbericht gespeichert. Danach werden alle allozierten Speicherbereiche freigegeben und die erstellten Objekte zerstört. Hierzu reicht es normalerweise aus, das Zerstören aller Singleton-Objekte durch den Singleton-Manager[3] zu veranlassen.

1 z.B. „Kein Fenster ist mehr offen“, „Zeit T ist seit Aufruf vergangen“ oder „N Bilder wurden bearbeitet“
2 Siehe auch den folgenden Abschnitt V.2.3 „Dynamisch aktualisieren: cl_UpdateManager“.
3 Siehe auch den Abschnitt V.2.5 „Geordnet aufräumen: cl_SingletonManager“.

V.2.3. Dynamisch aktualisieren: cl_UpdateManager

In den meisten Softwareanwendungen gibt es eine so genannte „Mainloop“. In dieser Schleife werden sequentiell alle notwendigen Algorithmen ausgeführt und alle Daten und Objekte aktualisiert. Dies müssen nicht dieselben Algorithmen, Daten und Objekte zur gesamten Laufzeit sein.

Beispielsweise ist denkbar, dass zunächst ein Algorithmus nach einem bestimmten Muster im Bild Ausschau hält und, wenn er es gefunden hat, auf einen zweiten Algorithmus umgeschaltet wird, welcher dann den Hubschrauber relativ zu dem Muster schweben lässt, bis eine Benutzereingabe auftritt, die auf einen dritten Algorithmus umschaltet, usw. Während der gesamten Zeit läuft parallel z.B. ein Algorithmus zur Kollisionsvermeidung und bei bestimmten Flugmanövern wird ein Algorithmus eingeschaltet, der die Kamerabilder aufzeichnet ... Eine Möglichkeit der Realisierung dieses Szenarios wäre eine große Schleife, in der für alle potentiellen Algorithmen und Objekte nacheinander entschieden wird, ob diese zur Zeit aktualisiert werden müssen oder momentan passiv sind. Dies hätte die Nachteile, dass erstens für jeden potentiellen Aktualisierungskandidaten in jedem Durchgang wenigstens etwas Rechenzeit aufgewendet und zweitens der Quellcode dieser Schleife für jedes weitere Update-Objekt, welches der Anwendung hinzugefügt wird, angepasst werden muss.

Die Klasse `cl_UpdateManager` aus dem Modul `util` löst diese Probleme, indem er eine interne Liste der momentan zu aktualisierenden Objekte führt und nur diese in jedem Durchgang sequentiell „updated“. Dieser Liste können zur Laufzeit dynamisch neue Update-Objekte hinzugefügt werden. Objekte, die nicht mehr zu aktualisieren sind, werden dagegen entfernt[1]. Update-Objekte müssen nur eine Voraussetzung erfüllen: sie müssen die Schnittstelle `i_UpdateObject` implementieren, d.h. sie müssen eine Methode `update()` besitzen. Diese wird dann in jedem Durchgang vom `cl_UpdateManager` aufgerufen um die notwendigen Aktualisierungen für das jeweilige Objekt vorzunehmen.

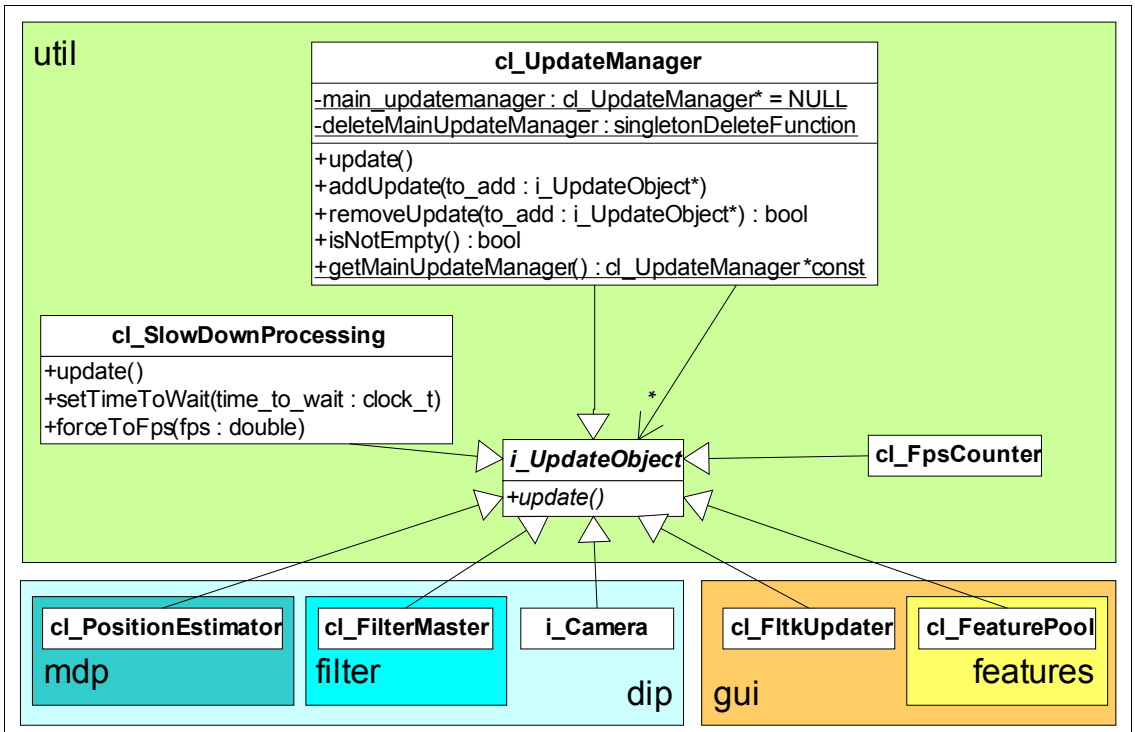
Eine typische „Mainloop“ einer Applikation, die mit Hilfe des dip-Frameworks implementiert wurde, enthält damit nur noch einen einzigen Aufruf, nämlich den der Methode `update()` des „`main_updatemanager`“. Dieser „`main_updatemanager`“ ist ein nach dem Singleton-Muster[2] implementiertes

1 Häufig entfernen sie sich nach Erledigung ihrer Aufgabe oder bei Destruktion sogar selbst.

2 In Abschnitt V.2.5 „Geordnet aufräumen: cl_SingletonManager“ wird das Singleton-Muster und seine Wirkungsweise näher beschrieben.

Objekt der Klasse `cl_UpdateManager` und damit global von allen Klassen und Funktionen der Anwendung erreichbar, welche je nach Situation und Aufgabe `i_UpdateObject`-Objekte bei ihm an- und abmelden[1].

Das UML-Diagramm V.2 zeigt die Klasse `cl_UpdateManager`, die Schnittstelle `i_UpdateObject` und alle derzeit abgeleiteten Klassen[2].



UML-Diagramm V.2: Die Klasse `cl_UpdateManager` *und Update-Objekte*

Ein `i_UpdateObject` ist z.B. die Klasse `cl_SlowDownProcessing`. Ihre Objekte können benutzt werden, um die Anwendung zu Simulations- oder Debugzwecken zu bremsen. Der Anwender oder andere Objekte können dabei eine gewisse Zeitspanne festlegen, die die Applikation in jedem Schritt „warten“ soll, oder vorgeben, dass in jedem Schritt jeweils so lange zu warten ist, dass eine gewisse „Framerate“ (Schritte pro Sekunde) nicht überschritten wird. Ein solches `cl_SlowDownProcessing`-Objekt prüft also ggf. bei jedem Aufruf von seiner `update()`-Methode durch den `cl_UpdateManager`, wie viel Zeit seit dem letzten `update()`-Aufruf vergangen ist, und lässt die Applikation dann die entsprechende Zeit warten.

---

1 Das An- und Abmelden geschieht durch Aufruf der Methoden `addUpdate()` und `removeUpdate()`.

2 Für eine bessere Übersichtlichkeit wurden nicht alle Methoden und Attribute der Klassen dargestellt. Zur Erläuterung der verwendeten Symbolik siehe auch Anhang VII.1.

Eine Besonderheit der Klasse `cl_UpdateManager` ist, dass sie (wie im UML-Diagramm V.2 zu sehen) selber die Schnittstelle `i_UpdateObject` implementiert. Dadurch ist es möglich, dass man mehrere `cl_UpdateManager`-Objekte ineinander verschachteln kann. Man kann z.B. alle `i_UpdateObject`-Objekte, die zum Erfüllen einer bestimmten Aufgabe notwendig sind, in einem `cl_UpdateManager`-Objekt zusammenfassen und braucht nur noch jenes dem „`main_updatemanager`“ hinzufügen, wenn die Aufgabe erfüllt werden soll, und entfernen, wenn die Aufgabe erledigt ist.

V.2.4. Messen der Verarbeitungszeit: cl_FpsCounter

Wie schon erwähnt (z.B. in Abschnitt I.1) ist die Zeit bei der Bildverarbeitung von Bildsequenzen eine kritische Größe. Daher ist es insbesondere beim Entwickeln neuer und Testen vorhandener Algorithmen wichtig, über deren Geschwindigkeit Auskunft zu erhalten. Auch für Algorithmen kann das Wissen um die aktuelle Framerate (wieviele Bilder in der Sekunde bearbeitet werden können) nützlich sein, entweder um daran ihre Genauigkeit bzw. Geschwindigkeit auszurichten oder auch um die aktuelle Geschwindigkeit z.B. des Helikopters zu ermitteln.

Die Objekte der Klasse `cl_FpsCounter` im Modul `util` zählen die Aufrufe ihrer Methode `update()`[1] und messen jeweils die Zeit, die zwischen zwei `update()`-Aufruf vergangen ist. Daraus ermitteln sie die aktuelle, die durchschnittliche, die maximale und die minimale Framerate (seit ihrer Intialisierung bzw. dem letzten Aufruf von `reset()`).

Wie die Klasse `cl_UpdateManager` bietet auch cl_FpsCounter ein Singleton[2]-Objekt an. Dieses wird nach seiner Erstellung beim „`main_updatemanager`“[3] angemeldet und misst somit die benötigte Zeit für einen „Mainloop“-Durchgang, was in der Regel die Bearbeitungszeit für ein Bild ist.

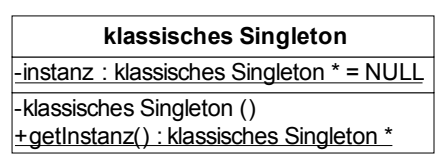
1 Auch sie implementiert die Schnittstelle `i_UpdateObject`.

2 Siehe auch den nächsten Abschnitt V.2.5 „Geordnet aufräumen: cl_SingletonManager“

3 Singleton-Objekt der Klasse `cl_UpdateManager`, siehe vorheriger Abschnitt V.2.3 „Dynamisch aktualisieren: cl_UpdateManager“

V.2.5. Geordnet aufräumen: cl_SingletonManager

Das Singleton-Entwurfsmuster (singleton-pattern) ist eines der bekanntesten Entwurfsmuster. In seiner ursprünglichen Form (wie vorgestellt in [16]) wird es verwendet, um Klassen, von denen höchstens ein Objekt existieren soll[1] und die von allen anderen Klassen und Funktionen gleichermaßen einfach zugänglich sein sollen, zu implementieren. Für die Implementierungen in diesem Framework war vor allem letztere Eigenschaft (das Objekt ist von allen anderen Klassen gleichermaßen einfach zugänglich) interessant. Erstere (es gibt höchstens ein Exemplar), war meist unwichtig, für manche Klassen sogar unerwünscht.



UML-Diagramm V.3: Singleton-Klasse in ursprünglicher Form

Daher ist eine Singleton-Klasse im Sinne dieses Frameworks eine Klasse, die auch (neben anderen) ein einmaliges[2], allgemein zugängliches Objekt (in dieser Arbeit auch Singleton-Objekt genannt) haben kann. Das heißt, der Konstruktor kann öffentlich sein. Ansonsten ist die Implementierung zunächst einmal die gleiche wie im klassischen Singleton-Entwurfsmuster:

1. Es gibt als privates Klassenattribut einen Zeiger auf ein Klassenobjekt, dieser wird mit dem Null-Zeiger initalisiert.

2. Es gibt eine Klassenmethode, die beim ersten Aufruf ein neues Objekt der Klasse erzeugt, den Zeiger darauf zeigen lässt und in jedem Aufruf den Wert des Zeigers (also die Adresse des Singleton-Objektes) zurückgibt.

Wie und wann ein Singleton-Objekt erzeugt wird und wer dafür verantwortlich ist, ist also klar[3]. Ein in der Softwaretechnik häufig diskutiertes Problem ist aber die Frage: „Wer oder was zerstört wann ein Singleton-Objekt?“ [17][18][19]

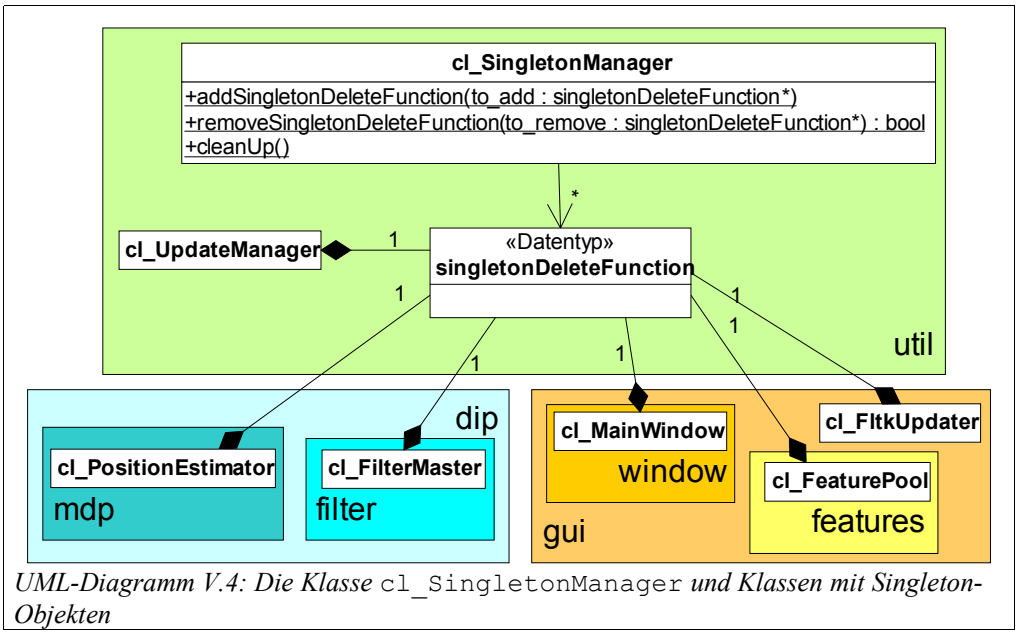
1 wie z.B. Speichermanager oder sonstige Ressourcenverwalter

2 einmalig, da es das einzige auf diese Weise konstruierte Objekt ist

3 Verantwortlich ist die genannte Klassenmethode bzw. derjenige, der sie zum ersten mal aufruft. Ein weiterer Vorteil des Singletons in dieser Form (es gibt auch andere Implementierungen, in denen die Instanz selbst das Klassenattribut darstellt) ist übrigens, dass das Singleton-Objekt erst erzeugt wird, wenn es wirklich (zum ersten Mal) gebraucht wird. Falls also z.B. durch einen Fehler beim Erstellen einer Anwendung (z.B. falsches/überflüssiges „#include“) die Klasse `cl_MainWindow` mit in die Anwendung kompiliert und gelinkt, aber ihre Methode `getMainWin()` nicht auf gerufen wird, so wird auch `main_win` nicht erzeugt, was ansonsten Geschwindigkeitseinbußen und erhöhten Speicherplatz-Verbrauch zur Folge hätte.

Gerade das „wann“ ist entscheidend, wenn es Abhängigkeiten zwischen Singleton-Objekten gibt, die erfordern, dass diese in einer bestimmten Reihenfolge zerstört werden[1].

Für dieses Problem gibt es verschiedene Lösungsansätze[2]. Im dip-Framework wurde eine einfache Variante des Singleton-Managers [17] gewählt. Unser Ansatz geht davon aus, dass die am Ende der Ausführung existierenden Singleton-Objekte in der umgekehrten Reihenfolge ihrer Konstruktion zerstört werden sollen[3]. Alle Singleton-Objekte bzw. die Funktionen, die diese jeweils zerstören, werden zum Zeitpunkt der Konstruktion der Singleton-Objekte durch die entsprechende Klassenmethode beim `cl_SingletonManager` angemeldet.



UML-Diagramm V.4: Die Klasse `cl_SingletonManager` *und Klassen mit Singleton-Objekten*

1 Wenn z.B. der Destruktor eines Singleton-Objektes die `getInstanz()`-Klassenmethode eines zweiten zuvor bereits zerstörten Singleton-Objektes aufrufen würde, so würde dieses, neu konstruiert werden, was mindestens unschön, unter Umständen aber auch fatal ist, wenn nämlich das erste Singleton-Objekt erwartet, dass sich das zweite in einem bestimmten Zustand befindet, den es nach der Neu-Konstruktion nicht mehr hätte.

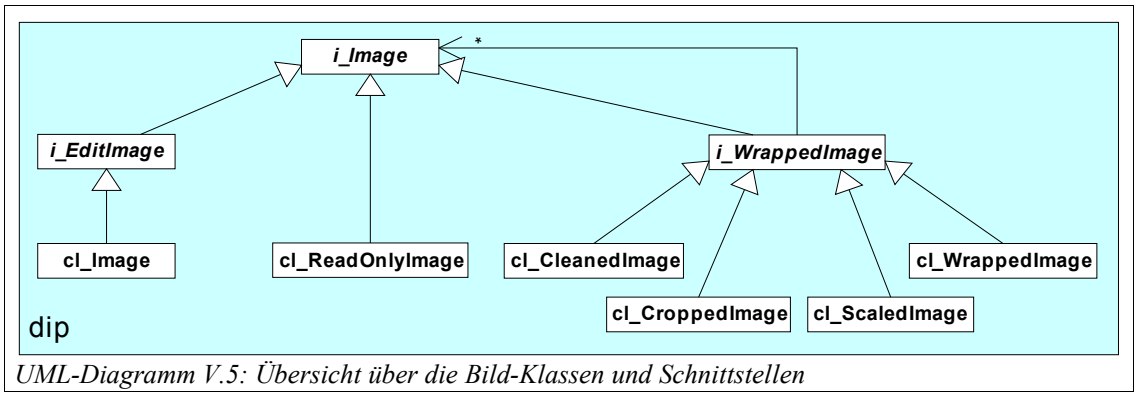
2 Lösungsansätze wie Referenzzähler (Singleton-Objekte werden zerstört, wenn niemand mehr Interesse daran zeigt. Im Falle des dip-Framework wurde dagegen entschieden, weil es sinnvoll sein kann, dass z.B. ein main_fps_counter auch dann noch weiter existiert, um zum Ende der Anwendung nochmals auf ihn zurückzugreifen, und seine über die gesamte Zeit gemessenen Werte auszugeben), die Haltung „Das Betriebssystem gibt den Speicher schon frei, wenn die Anwendung beendet wurde!“ (Dies muss erstens nicht notwendigerweise der Fall sein und zweitens kommt so kein Destruktoraufruf für die Singleton-Objekte zu Stande, was unangenehme Folgen haben kann, falls dadurch z.B. abschliessende Ausgaben nicht erfolgen oder Log-Dateien finalisiert werden) u.a.

3 Dies ist in fast allen Fällen praktikabel.

Die `main()`-Funktion ruft dann kurz vor Ende der Anwendung die Methode `cleanUp()`[1] des `cl_SingletonManager` und dieser die gespeicherten Methoden in ihrer umgekehrten Reihefolge auf, so dass alle Singleton-Objekte zerstört werden. Wird ein Singleton-Objekt aus irgendeinem Grund schon vor Aufruf von `cleanUp()` zerstört, so muss die entsprechende Methode natürlich auch aus dem `cl_SingletonManager` entfernt werden.

V.3. Die Bild-Klassen und -Schnittstellen

Das zentrale Objekt der digitalen Bildverarbeitung ist, wie der Name schon sagt, das Bild. Es gibt im dip-Framework unterschiedliche Arten von Bildern. Solche, die editiert werden können, und solche, deren Daten nur gelesen werden dürfen. Außerdem gibt es noch welche, die keine eigenständigen Bilder darstellen, sondern letztendlich auf anderen Bildern beruhen, weil sie z.B. ein Ausschnitt von einem anderen Bild sind.



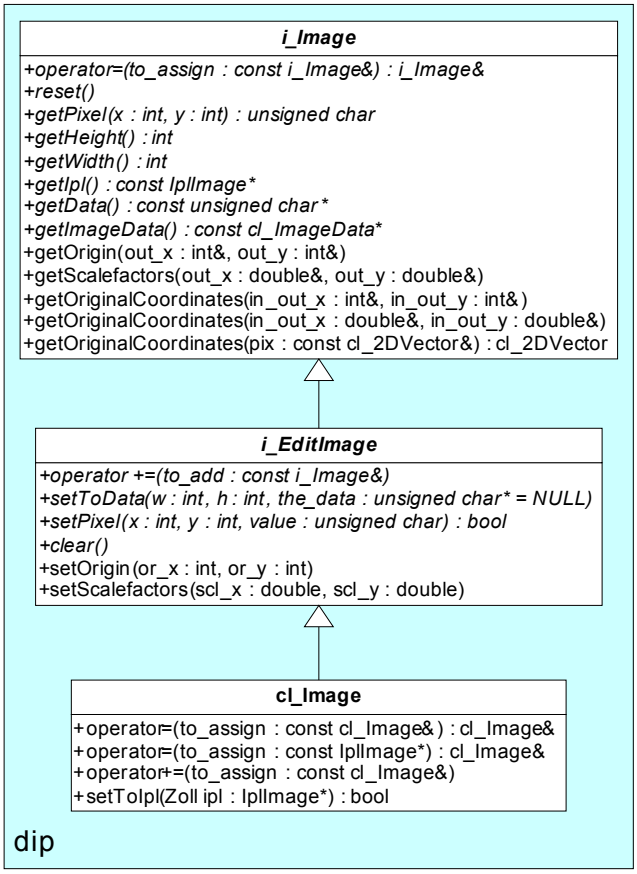
UML-Diagramm V.5: Übersicht über die Bild-Klassen und Schnittstellen

All diese Bildobjekte implementieren letztendlich die zentrale Schnittstelle des Frameworks `i_Image`. Sie und ihre Kindschnittstellen und -klassen werden im UML-Diagramm V.5 und den folgenden Abschnitten vorgestellt.

V.3.1. Veränderbare Graustufenbilder

Die Schnittstelle `i_Image` bietet die grundlegenden Informationen über ein Bild. Das sind natürlich in erster Linie die einzelnen Pixelwerte oder der Zeiger auf das Array, in dem diese gespeichert sind. Aber auch die Dimensionen des Bildes (Breite und Höhe) gehören dazu, genauso wie seine Position relativ zu

1 Siehe auch Abschnitt V.2.2 „Anfang und Ende der Applikationsausführung: main()“, insbesondere Codebeispiel V.1.



UML-Diagramm V.6: Die Klasse `cl_Image` *und von ihr implementierte Schnittstellen.*

anderen Bildern oder auch Skalierungsfaktoren, um das Bild zu anderen in Beziehung setzen können. Letztere sind vor allem wichtig, um die Pixelkoordinaten von Ausschnitten und Skalierungen auf die des Original-Kamerabildes beziehen zu können. Das wiederum ist notwendig, um z.B. letztendlich[1] Aussagen über die Position des Gesehenen in der Welt machen zu können. Um dies zu erleichtern, wurden Methoden zur Verfügung gestellt, die gegebene bildrelative Koordinaten in Koordinaten des Originalbildes umwandeln.

Zusätzlich gibt es noch eine Methode `getIpl()`, die das Bild im OpenCV-Format `IplImage` ausgibt, einen Zuweisungsoperator, der das Bild zur Kopie eines anderen werden lässt, und eine Methode `reset()`, die das Bild löscht[2].

Die Schnittstelle `i_EditImage` ist für editierbare Bilder gedacht. Bei diesen Bildern können die einzelnen Pixelwerte (`setPixel()`) oder auch der gesamte Datenbereich neu gesetzt werden. Die Methode `clear()` z.B. setzt alle Pixelwerte auf Null (schwarz).

Die Methode `setToData()` hingegen setzt für das gesamte Bild einen neuen Datenbereich und auch die Dimensionierung kann hier angepasst werden.

1 nach Einbeziehung von Kenntnissen über die Kamera, den Pixelabständen usw.

2 D.h. das Objekt repräsentiert hinterher ein Bild mit der Ausdehnung 0 x 0. Das Objekt selbst wird also nicht zerstört.

Außerdem kann man die Angaben über die relative Position bzw. die Skalierungsfaktoren neu setzen, falls die auf das Bild angewendeten Algorithmen etwas daran geändert haben sollten.

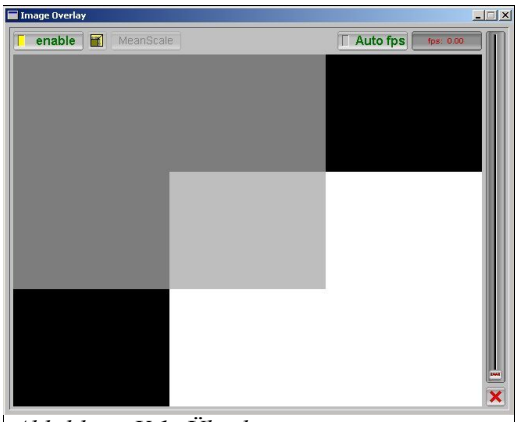


Abbildung V.1: Überlagerung von gegeneinander verschobenen Grauwertbildern

Mit dem „+="-Operator kann ein anderes Bild dem `i_EditImage` überlagert werden. Das ist auch mit Bildern möglich, die nicht deckungsgleich sind (siehe auch Abbildung V.1). In Bildbereichen, die sich überschneiden, werden als neue Pixelwerte die arithmetischen Mittel der jeweiligen Pixelwerte der beiden Ausgangsbilder gesetzt. In allen anderen Bereichen werden die Pixelwerte des jeweiligen Bildes kopiert, bzw. schwarz gesetzt.

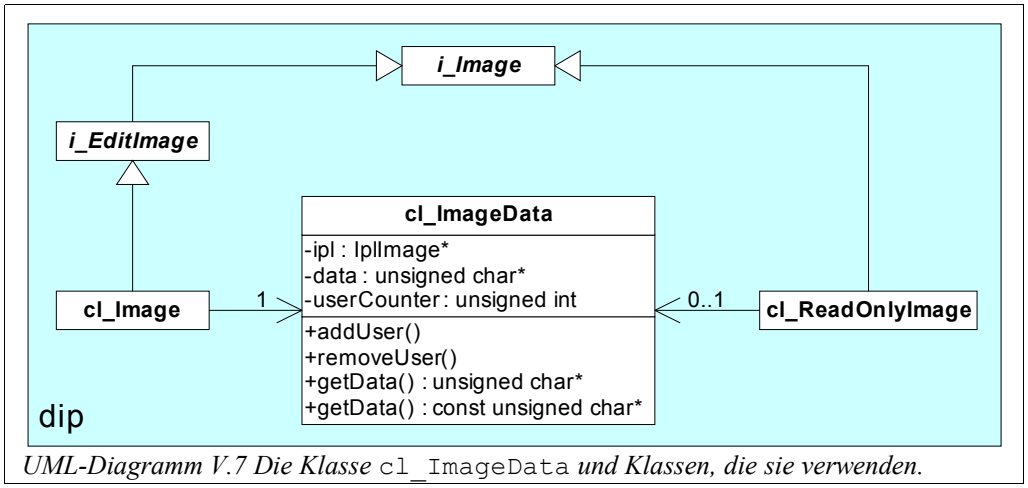
`cl_Image` ist eine (und zur Zeit die einzige) Implementierung von `i_EditImage`. `cl_Image` hat neben den geerbten noch zusätzliche Methoden, die das Umwandeln von 3- und 1-Kanalbilder im `IplImage`-Format in (Graustufen-) `cl_Image`-Objekte unterstützen. Darüber hinaus sind überladene Operatoren und Konstruktoren implementiert worden, die zum Kopieren, Überlagern und Zuweisen von `cl_Image`s optimierte Algorithmen verwenden.

V.3.2. Unveränderbare Bildkopien

Manche Filter und Algorithmen müssen ein Eingangsbild über mehrere Schritte zur Verfügung haben, um es beispielsweise mit späteren Bildern vergleichen zu können. Sie wollen das Bild dabei nicht verändern, sondern es nur „anschauen", nur lesend auf die Daten zugreifen. Die anderen Filter oder die Kamera, von der das Bild empfangen wird, ändern aber erstens ihre Ausgabebilder ständig und sind zweitens als Besitzer bzw. „Ersteller" der Bilder auch diejenigen, die dafür verantwortlich wären, den von ihnen reservierten Speicher wieder freizugeben. Diese können aber nicht „wissen", wann die Bilddaten nicht mehr benötigt werden[1]. Die einfachste Lösung wäre sicherlich, dass Filter, die ein fremdes Bild

1 Man könnte zwar ermitteln, wie viele Schritte lang ein Bild zur Zeit maximal für alle implementierten Filter aufbewahrt werden müsste, dies führt aber insgesamt zu zu hohem Speicherverbrauch (u.a. da nicht immer alle Filter im Einsatz sind).

über einen Schritt hinaus benötigen, sich eine Kopie davon anfertigen. Da die Bilddatenmenge aber in der Regel recht groß ist, dauert das Kopieren dem entsprechend lange. Um solche Zeitverluste nicht in Kauf nehmen zu müssen, wurde ein anderer Lösungsansatz gewählt.



UML-Diagramm V.7 Die Klasse `cl_ImageData` *und Klassen, die sie verwenden.*

Es wurde eine eigene Klasse für reine Bilddaten[1] (`cl_ImageData`) erstellt. Diese besitzt einen Zähler, der angibt, wie viele Objekte, Funktionen oder ähnliches sich gerade für den Datenbereich „interessieren". Andere Objekte können den Zähler durch Aufruf der Methode `addUser()` bzw. `removeUser()` erhöhen bzw. erniedrigen. Sinkt dabei der Zähler auf Null, d.h. es ist kein Objekt mehr an dem Datenbereich interessiert, so wird dieser automatisch zerstört (siehe auch UML-Diagramm V.7). Um Zugriffskonflikte zu vermeiden, darf auch weiterhin nur eine Klasse (im Normalfall diejenige, welche den Datenbereich erstellt hat) auf den Datenbereich schreibend zugreifen. Allerdings sollte sie ihn nach „Veröffentlichung", also nachdem sie ihn einem anderen Bild zur Verfügung gestellt hat, nicht mehr ändern. Daher erzeugen die meisten Filter und andere Klassen, wie z.B. die Kameras, die eigene Bilder produzieren, bei jedem Schritt einen neuen Datenbereich[2].

Filter, die also ein Bild einige Zeit aufbewahren müssen, erzeugen sich nunmehr keine echte Kopie, sondern sie verwenden die zusätzliche Bildklasse `cl_ReadOnlyImage`. Diese repräsentiert, wie ihr Name vermuten lässt, Bilder auf deren Daten nur lesend zugegriffen werden darf. Wenn nun ein anderes Bild einem `cl_ReadOnlyImage`-Objekt mittels Operator- oder Konstruktoraufruf zugewiesen wird, so kopiert es sich nur die Metadaten zu dem Bild (wie Höhe,

1 ohne Zusatzinformation wie Breite, Höhe o.ä., allerdings mit dem zugehörigen `IplImage`, falls die Daten aus einem solchen stammen, damit sie korrekt zerstört werden können.

2 Dies ist zeitlich gesehen nicht weiter tragisch, da sie normalerweise sowieso den gesamten Bildbereich überschreiben würden.

```
dip::i_Image& dip::cl_ReadOnlyImage::operator=(const i_Image& to_assign)
{
   if (&to_assign!=this)
   {
      if (data)
         data->removeUser();
      data=to_assign.getImageData();
      if(data)
         data->addUser();
      width=to_assign.getWidth();
      height=to_assign.getHeight();
      to_assign.getOrigin(origin_x,origin_y);
      to_assign.getScalefactors(scale_x,scale_y);
   }
   return *this;
}
```

Codebeispiel V.2: Implementierung des Zuweisungsoperators der Klasse `cl_ReadOnlyImage`

Breite usw.) und merkt sich das zugehörige Bilddaten- (`cl_ImageData`-) Objekt und erhöht dessen Benutzerzähler (siehe Codebeispiel V.2). Sobald es ein anderes Bild zugewiesen bekommt oder zerstört wird, dekrementiert es jenen Benutzerzähler wieder. D.h. die „Lebensdauer“ der Bilddaten im Speicher wird nun vollkommen von der Art der Filter, die es benutzen, also von der Zeit bestimmt, in der diese Daten tatsächlich gebraucht werden.

V.3.3. Maskierte Bilder

Bei `i_WrappedImage`-Objekten handelt es sich um Bilder, die auf anderen Bildobjekten basieren. Diese anderen Bilder werden von den `i_Wrapped`-Image-Objekten „eingepackt“, maskiert. So ist von dem eigentlichen Bild bei Zugriff über ein `i_WrappedImage` nur ein Ausschnitt zu sehen (`cl_CroppedImage`), die Vergrößerung bzw. Verkleinerung (`cl_ScaledImage`) oder ein Bild, bei dem Pixelwerte[1] geglättet wurden. Somit ist ein `i_WrappedImage` prinzipiell auch ein Filter. Das Besondere ist hierbei, dass die entsprechenden Algorithmen erst angewendet werden, sobald auf die entsprechenden Pixel zugegriffen wird. Dieser Mechanismus sei hier beispielhaft an der Klasse `cl_CroppedImage`, die auch vom Typ `i_WrappedImage` ist, erläutert:

Gegeben sei ein Bild (`i_Image`) der Größe 100 x 100 Pixel. Dieses Bild wird per `setInnerImage()` einem `cl_CroppedImage`-Objekt hinzugefügt. Es wird

1 Pixel deren Wert höher bzw. niedriger war als der aller Nachbarpixel

ein Ausschnitt von 30 x 40 Pixeln (`setSize()`) an der Position 10,10 (`setPosition()`) gewählt. Für ein anderes Objekt stellt dieses `cl_CroppedImage` nun nach außen ein Bild der Größe 30 x 40 dar. Greift ein anderes Objekt nun z.B. auf Pixel 5,3 des „neuen" Bildes zu, so berechnet das `cl_CroppedImage`-Objekt zunächst dessen Koordinaten im Ausgangsbild (15,13) und liefert den entsprechenden Pixelwert an dieser Stelle zurück (siehe auch Codebeispiel V.3).

```
uchar dip::cl_CroppedImage::getPixel(int x_pos, int y_pos)const
{
   if (x_pos<w&&y_pos<h)
   {
      return inner_image->getPixel(x_pos+x,y_pos+y);
   }
   return 0;
}
```

Codebeispiel V.3: Die Methode `getPixel()` *von* `cl_CroppedImage`

Man kann `i_WrappedImage`-Objekte (da auch sie vom Typ `i_Image` sind) natürlich auch in einander schachteln. Der Vorteil dieser Objekte bleibt stets, dass die Algorithmen nur angewendet werden, wenn es für das jeweilige Pixel auch notwendig ist.[1]

V.4. Gespeicherte Blobeigenschaften

Im dip-Framework werden Blobs durch die Klasse `cl_Blob` (im Modul `dip::blob`) repräsentiert. Wie schon in Abschnitt III.2 besprochen, ist es vorteilhaft nur die Eigenschaften eines Blobs zu speichern und nicht alle enthaltenen Pixel. Die zu jedem Blob intern abgelegten Eigenschaften wurden nach drei Gesichtspunkten ausgewählt:

1. Es muss möglich sein, ohne erneutes Betrachten aller schon erfassten zugehörigen Pixel, einem Objekt vom Typ `cl_Blob`, einen weiteren Pixel oder anderen Blob hinzuzufügen. Hierdurch wird schnelles „Sammeln"[2] von Blobs ermöglicht.

1 Damit keine Speicherzugriffsfehler entstehen, falls das innere Objekt zerstört wird, wurde übrigens ein Mechanismus implementiert, über den jedes `i_Image` die `i_WrappedImage`-Objekte, in denen es aktuell „eingepackt" ist, bei seiner Zerstörung benachrichtigt.

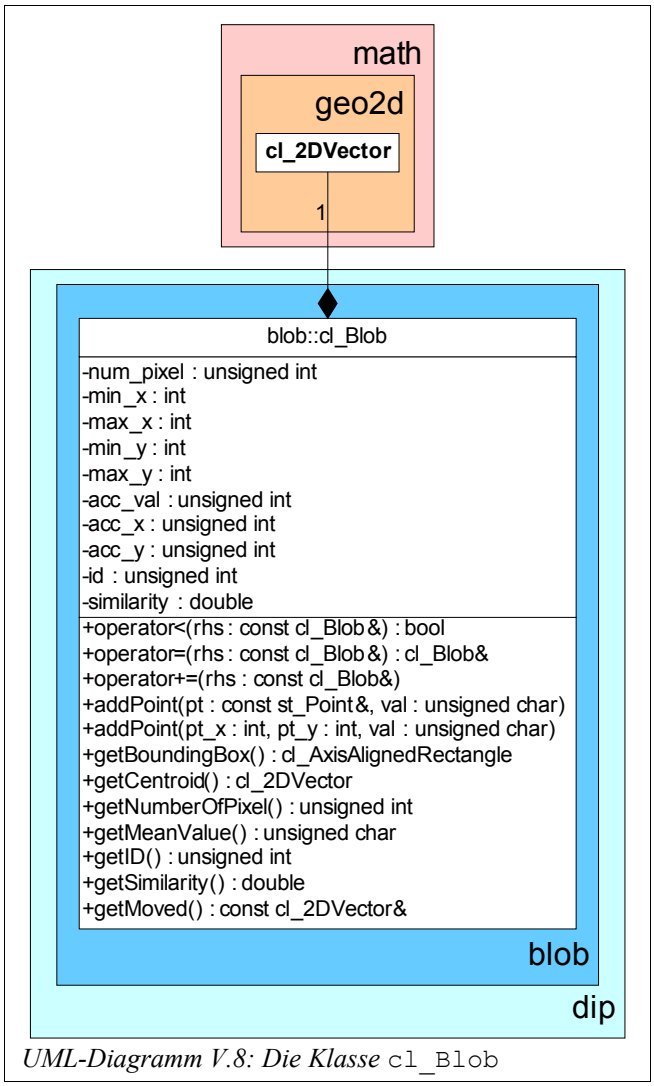
2 „Sammeln" meint das Heraussuchen aller in einem Bild vorhandenen Blobs, siehe auch Abschnitt III.4 „Algorithmen zum „Sammeln" der Blobs".

2. Eigenschaften, die sich einfach aus anderen ableiten lassen, müssen nicht gespeichert werden.

3. Die Eigenschaften müssen von implementierten Filtern und Algorithmen, direkt oder indirekt[1] verwendet werden.

Aus Letzterem folgt, dass die Menge der gespeicherten Eigenschaften für zukünftige Algorithmen eventuell erweitert werden muss. So wird z.B. momentan noch nicht der Umriss eines Blobs in `cl_Blob` abgelegt, zur besseren Identifizierung kann das aber durchaus sinnvoll sein.

Zur Zeit werden folgende Eigenschaften in einem `cl_Blob`-Objekt gespeichert:



UML-Diagramm V.8: Die Klasse `cl_Blob`

- Anzahl der enthaltenen Pixel (`num_pixel`).

- Jeweils die kleinste und die größte x- und y-Bildkoordinate der enthaltenen Pixel (`min_x`, `min_y`, `max_x`, `max_y`).

- Die Summe aller Pixelwerte (`acc_val`).

- Jeweils die Summe der x- und y-Bildkoordinaten aller enthaltenen Pixel (`acc_x`, `acc_y`).

1 Durch Nutzen einer abgeleiteten Eigenschaft

- Eine ID, die bei der Wiederfindung, der Identifizierung einem Blob zugeordnet wird[1] (`id`).

- Ein Wert, der die Ähnlichkeit eines identifizierten Blobs mit dem ihm zugeordneten Blob aus dem vorherigen Bild angibt (`similarity`).

- Ein Vektor, der von der Position[2] eines identifizierten Blobs im letzten Bild zu seiner jetzigen Position zeigt, der also die letzte Bewegung des Blobs angibt (`moved`).

Aus diesen Basiseigenschaften können bei Bedarf folgende, weitere Eigenschaften eines Blobs abgeleitet werden:

- Die Bildkoordinaten des Schwerpunktes eines Blobs (ergibt sich jeweils aus der Summe der x- bzw. y-Bildkoordinaten aller enthaltenen Pixel geteilt durch die Anzahl der enthaltenen Pixel).

- Das kleinste, umschließende, achsenparallele[3] Rechteck und damit natürlich auch dessen Ausdehnung in x- und y-Richtung, sowie dessen Mittelpunkt. (Die Eckpunkte des Rechtecks ergeben sich aus der jeweils kleinsten und größten x- bzw. y-Bildkoordinate der enthaltenen Pixel)

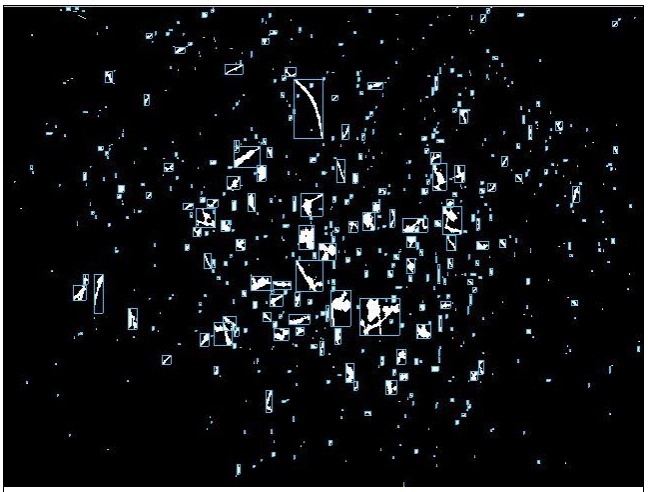


Abbildung V.2 Blobs jeweils mit eingezeichneter "axis-aligned bounding-box".

- Den durchschnittlichen Pixelwert (die durchschnittliche Helligkeit) des Blobs (ergibt sich aus der Summe aller Pixelwerte geteilt durch die Anzahl der Pixel).

1 Näheres siehe Abschnitt III.6 „Blob-Identifizierung“

2 Als „Position“ des Blobs wurde stets sein Schwerpunkt angenommen.

3 Auch unter dem englischen Ausdruck „axis-aligned bounding-box“ bekannt. Die Seiten des Rechtecks sind jeweils parallel zu einer Achse des Bildkoordinatensystems.

V.5. Die Filterklassen und ihre dynamische Anordnung

Die Filter bearbeiten Bilder, verändern, verstärken, reduzieren oder extrahieren Informationen in ihnen, sind also neben den Bildern die wichtigste Gruppe von Objekten in der Bildverarbeitung. Wie in Abschnitt V.1.1 schon erwähnt, ist es besonders vorteilhaft, die Filter als Objekte und nicht etwa als Funktionen zu modellieren, da man so die Parametrisierung einfach von der Ausführung der Filteralgorithmen trennen kann, was wiederum zu einer einheitlichen Schnittstelle für alle Filter führt. Diese Schnittstelle erleichtert es letztendlich erheblich, die Reihenfolge, Zahl und Art der Filter zur Laufzeit bestimmen und ändern zu können. Im dip-Framework heißt die Filterklassen-Schnittstelle `i_Filter`. `cl_FilterMaster` ist der Name der Klasse von Objekten, mittels derer `i_Filter`-Objekte erzeugt und angeordnet werden können.

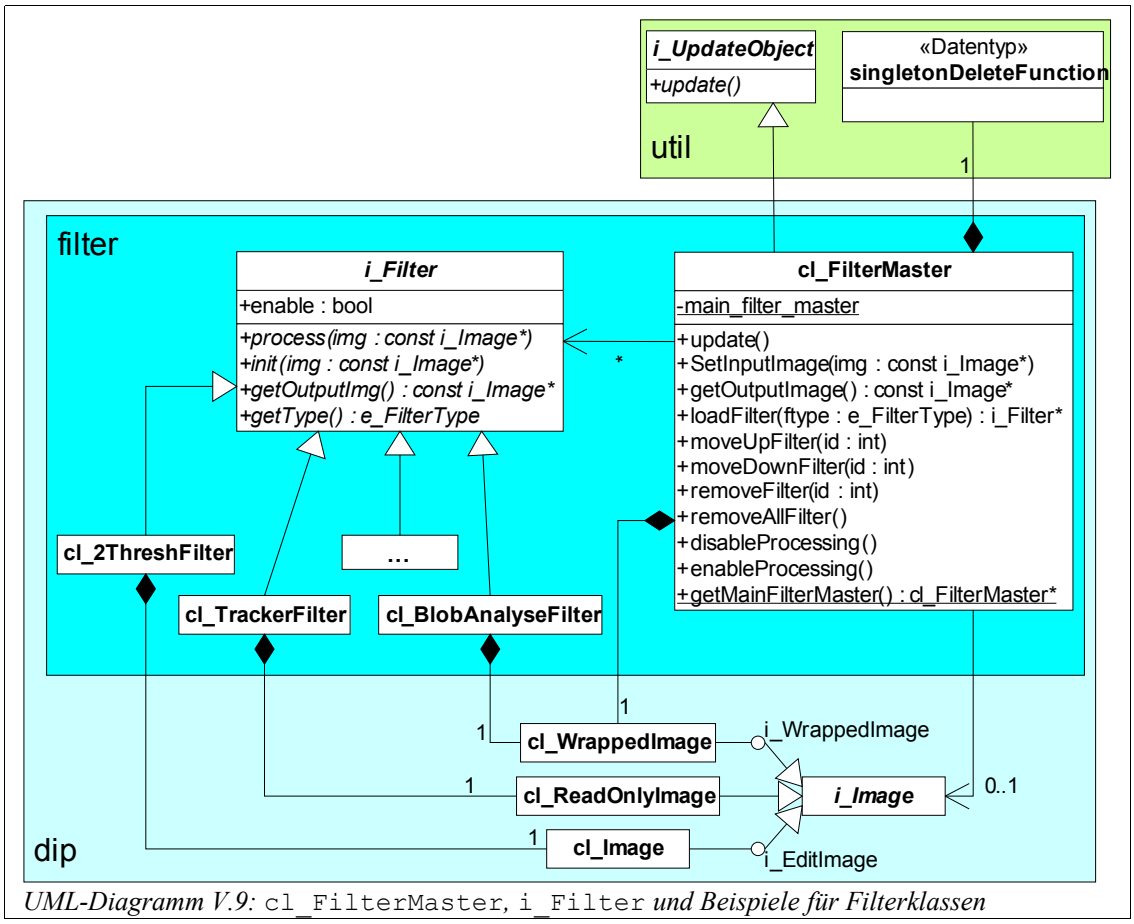
`i_Filter` wurde wie auch `cl_FilterMaster` von Olaf Guth definiert und implementiert. Da aber natürlich auch die Filter, die für diese Arbeit neu implementiert wurden, vom Typ `i_Filter` sind, werden beide an dieser Stelle kurz erläutert[1].[3]

Die Methode von `i_Filter`, die Bilder letztendlich verarbeitet, ist `process()`. Ihr ist das zu bearbeitende Bild zu übergeben, welches sie (da es in der Regel nicht von dem Filterobjekt erstellt wurde) nicht verändern darf. Ein Filter muss mittels der Methode `getOutputImage()` ein Ausgabebild zur Verfügung stellen, welches entweder das Ergebnis der Bearbeitung oder das „durchgeschleifte“ Eingangsbild sein kann.[2] Neben einer Methode, die Auskunft über den tatsächlichen Typ des Filters gibt (`getType()`), stellt `i_Filter` noch eine Methode `init()` zur Verfügung, die dazu gedacht ist, die Filter zu initialisieren (einige Filter benötigen ein erstes Bild) bzw. die Filter auf andere Bedingungen neu einzustellen. Die Variable `enable` zeigt einem `cl_FilterMaster`-Objekt an, ob ein Filter aktiv ist. Ist ihr Wert `false`, wird der Filter bei der Bearbeitung übersprungen.

In der Klasse `cl_FilterMaster` können Filter erstellt (`loadFilter()`) und beliebig, auch zur Laufzeit, neu angeordnet werden. In jedem Aufruf der

1 Die konkrete `i_Filter`-Implementierung `cl_BlobAnalyseFilter` wird im folgenden Abschnitt V.6 vorgestellt.

2 Das „Durchschleifen“ ist am besten mit Objekten der Klasse `cl_WrappedImage` zu bewerkstelligen, da diese ein Bild in keiner Weise verändern oder kopieren, wobei ihre Adressen natürlich nicht verändert werden, wenn das „innere“ Bild wechselt.



UML-Diagramm V.9: `cl_FilterMaster`, `i_Filter` *und Beispiele für Filterklassen*

Methode `update()`[1] eines `cl_FilterMaster`-Objektes wird – in der aktuellen Reihenfolge – jeweils die Methode `process()` der aktuell beinhalteten Filter ausgeführt, wobei dem ersten Filter das Eingangsbild des `cl_FilterMaster`-Objektes übergeben wird. Alle folgenden Filter erhalten das Ausgabebild des jeweils vorherigen Filters. Das Ausgabebild des `cl_FilterMaster` entspricht dem Ausgabebild des letzten Filters.

Die Reihenfolge der Filter lässt sich durch die Methoden `moveUpFilter()` bzw. `moveDownFilter()` verändern. Dadurch wird der angesprochene Filter in der aktuellen Bearbeitungsreihenfolge um einen Platz nach vorn (`moveUpFilter()`) oder nach hinten (`moveDownFilter()`) verschoben. Hierzu wie auch zum Entfernen eines Filters muss die aktuelle Position (gennant „`id`“) des Filters angegeben werden. Diese Position muss das aufrufende Objekt zur Zeit noch kennen (z.B. durch Buchführen über die einzelnen Positionen). In Zukunft werden mit Sicherheit auch Methoden zur Verfügung gestellt werden, die diese

1 Der Filtermaster erbt von `util::i_UpdateObject`.

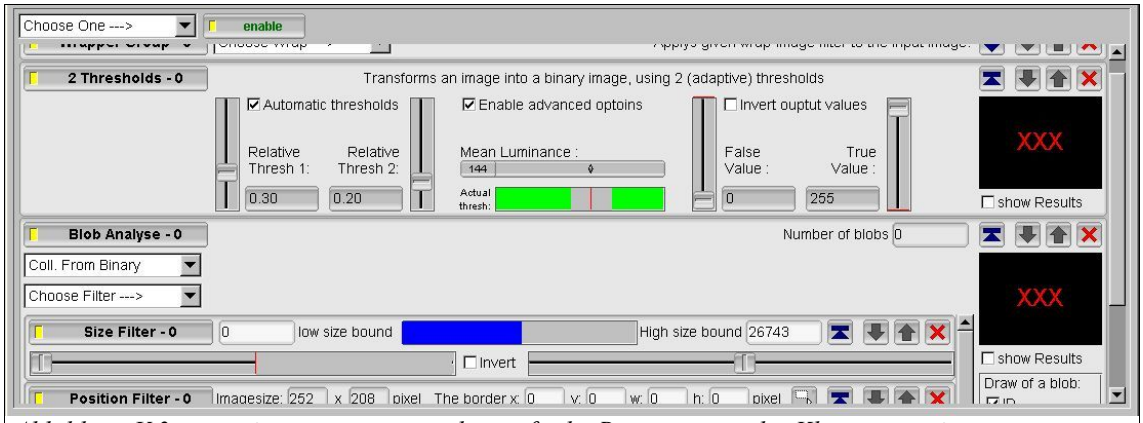


Abbildung V.3: `cl_FilterGroup` *ist die grafische Repräsentanz der Klasse* `cl_FilterMaster` *in einer GUI (z.B. in SPICE)*

Funktionen bei Übergabe eines `i_Filter`-Zeigers für den entsprechenden Filter ausführen[1].

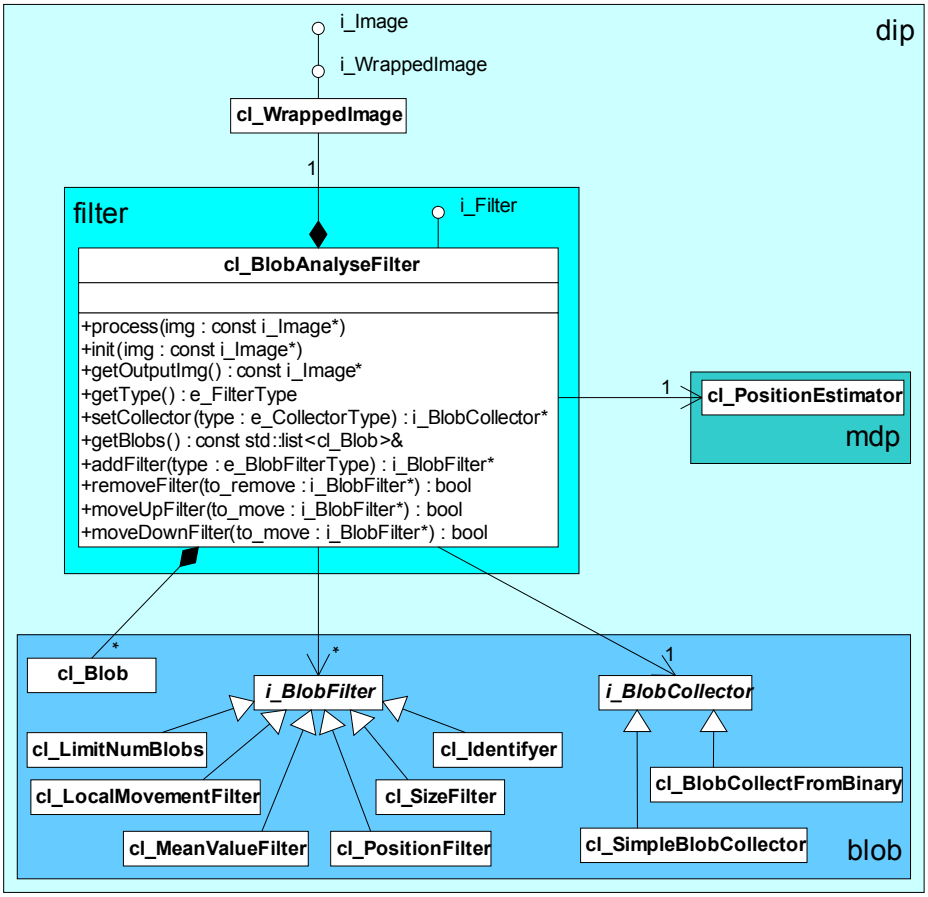
Abbildung V.3 zeigt die grafische Repräsentanz eines `cl_FilterMaster`-Objektes in einer GUI (`cl_FilterGroup`) inklusive der Darstellung einiger geladener Filter. Mit der Auswahlbox oben links (

) kann der Benutzer neue Filter hinzufügen. Der Button daneben dient zum Ein- und Ausschalten der Bearbeitung von Bildern (ruft `enableProcessing()` bzw. `disableProcessing()` auf). Mit Hilfe der drei quadratischen Buttons in der Repräsentanz eines jeden Filters oben rechts () kann man die Reihenfolge der Filter verändern und sie wieder löschen.

V.6. Die Filterklasse zur Blob-Analyse

Zur Realisierung der Blob-Analyse[2] wurde eine Filterklasse (`cl_BlobAnalyseFilter` im Modul `dip::filter`) entwickelt, deren Objekte aus einem Eingangsbild[3] Blobs sammeln, versuchen in der Menge dieser Blobs schon bekannte wieder zu finden und schließlich die aktuellen Blob-Positionen und Blob-Positionsänderungen an die Algorithmen zur Schätzung der Eigenbewegung der Kamera weitergeben. Sie implementiert die Schnittstelle `i_Filter` und kann somit jedem `cl_FilterMaster`-Objekt hinzu gefügt werden[4].

1 Oder zumindest eine Funktion, die bei Übergabe eines `i_Filter`-Zeigers die aktuelle Position des Filters zurückgibt.
2 Siehe auch Kapitel III „Blob-Analyse“.
3 Das Eingangsbild ist auch das Ausgangsbild. Es wird also nicht verändert, nur Informationen werden extrahiert.
4 Zu `i_Filter` und `cl_FilterMaster` siehe auch den vorherigen Abschnitt V.5.



UML-Diagramm V.10: `cl_BlobAnalyseFilter` *und ein Überblick über verwendete Klassen*

Als Datenstruktur zur Speicherung der Blobs wurde eine verkettete Liste gewählt, denn das Hinzufügen, vor allem aber auch das Löschen einzelner Objekte (in den Blob-Filtern) aus einer Liste geht schneller als beispielsweise bei einem `vector`-Objekt. Gefüllt wird diese Liste in jedem Schritt mit Hilfe eines Blobsammelalgorithmus. Für alle Blobsammelalgorithmen wurde eine eigene Schnittstelle definiert (`i_BlobCollector`, im Modul `dip::blob`). Die wichtige Methode dieser Schnittstelle ist `collect()`. Sie bekommt das entsprechende Eingangsbild des `cl_BlobAnalyseFilter`-Objektes und die verkettete Liste von Blobs übergeben. Die Liste wird zunächst geleert und danach werden alle im Bild gefundenen Blobs dort eingefügt.[1]

1 Zur Zeit gibt es zwei Klassen, die die Schnittstelle `i_BlobCollector` implementieren. Das ist zum einen die Klasse `cl_SimpleBlobCollector`, ihr Algorithmus wurde im Abschnitt III.4.1 „Blobs aus Graustufenbildern“ beschrieben. Die zweite Implementierung von `i_BlobCollector` ist auf Binärbilder optimiert. Die Klasse heißt `cl_BlobCollectFromBinary`. Der zugehörige Algorithmus und ein Verfahren zum Erstellen solcher Binärbilder wird in Abschnitt III.4.2 „Blobs aus Binärbildern“ vorgestellt.

Die Art und Zahl der geeigneten Blob-Filter sowie deren Reihenfolge kann – ähnlich den „Bildfiltern“ (`i_Filter`) – je nach Situation und Vorwissen unterschiedlich sein. Für Blob-Filter wurde deshalb eine eigene Schnittstelle (`i_BlobFilter`, im Modul `dip::blob`) eingeführt[1], deren Implementierungen den Objekten der Klasse `cl_BlobAnalyseFilter` – wie `i_Filter` den `cl_FilterMaster`-Objekten – hinzugefügt, angeordnet und gelöscht werden können. Aus Gründen der Flexibilität ist auch der Algorithmus zur Identifizierung von Blobs als ein `i_BlobFilter` implementiert worden, dies macht es möglich, auch nach der Identifizierung eine weitere Filterung der erhaltenen Blobs vorzunehmen. Wichtiger Unterschied zwischen `i_Filter`- und `i_BlobFilter`-Objekten ist allerdings – neben der Bearbeitung von Blob-Listen anstelle von Bildern – dass die Eingangs-Blob-Liste jeweils direkt von den Filtern verändert wird. Daraus folgen Geschwindigkeitsvorteile, da die ausgegebenen Blobs nicht kopiert oder neu erstellt werden müssen, aber auch der Nachteil, dass quasi parallele Bearbeitung derselben Liste durch mehrere Filter nicht ohne Weiteres (Anlegen von Kopien) möglich ist, da sonst Inkonsistenzen zu befürchten sind.

Der typische Ablauf bei Aufruf der Methode `process()` eines `cl_BlobAnalyseFilter`-Objektes zur letztendlichen Bestimmung der Eigenbewegung der Kamera ist also wie folgt (siehe auch Codebeispiel V.4):

1. Mittels eines `i_BlobCollector`-Objektes wird eine Liste der im Bild enthaltenen Blobs erstellt.

2. Aus der Liste werden mit Hilfe verschiedener `i_BlobFilter`-Objekte Blobs entfernt, so dass die Verbleibenden möglichst eindeutig identifizierbar sind.

3. Es wird versucht, die verbliebenen Blobs mittels eines `cl_Identifyer`-Objektes den Blobs des letzten Aufrufs zu zuordnen.

Ggf. werden die identifizierten Blobs nochmals gefiltert. Ihre Positionen und Bewegungsvektoren werden dann an die Algorithmen zur Schätzung der Eigenbewegung weitergegeben.

1 Näheres zu den Blob-Filter Algorithmen siehe Abschnitt III.5 „Filter für Blobs“

```
void dip::filter::cl_BlobAnalyseFilter::process(const dip::i_Image* img)
{
  if (img&&collector)
  {
     // Extracting the blob-data of the image.
     collector->collect(img,blobs);
     // Filtering (maybe incl. identifying) of the blobs.
     for(std::vector<blob::i_BlobFilter*>::iterator i=filters.begin();
                                                          i<filters.end();++i)
        if ((*i)->enable)
           (*i)->process(blobs);
     // Giving the found movements to the position estimator.
     double scl_x,scl_y;
     img->getScalefactors(scl_x,scl_y);
     math::geo2d::cl_2DVector tmp_v;
     for(std::list<blob::cl_Blob>::iterator ite=blobs.begin();ite!=blobs.end(); ++ite)
        if (ite->getID())
        {
           tmp_v=ite->getMoved();
           tmp_v.set(tmp_v.getX()*scl_x,tmp_v.getY()*scl_y);
           posesti->addMovement(mdp::cl_MeasuredMovement(
                                    img->getOriginalCoordinates(ite->getCentroid()),
                                                  tmp_v, ite->getSimilarity()));
        }
  }
  output.setInnerImage(img);
}
```

Codebeispiel V.4: Implementierung der Methode `process()` *der Klasse* `cl_BlobAnalyseFilter`

V.7. Gewinnung von zusätzlichen Debug-Informationen

Für den Benutzer ist es manchmal sinnvoll und wünschenswert zusätzliche Informationen und Daten visualisiert zu bekommen. Daten und Informationen, die zum Ausführen der eigentliche Algorithmen nicht vorgehalten werden müssen. Diese können dann z.B. zum Debuggen der Algorithmen verwendete werden, zur Demonstration ihrer Funktionsweise oder aber auch zur Bewertung dieser Algorithmen. Daher wurde z.B. der in Abschnitt IV.3 vorgestellte Algorithmus zweimal implementiert, in einer schnellen „normalen“ Variante und in einer „Debug“-Version, welche zahlreiche weitere Daten erzeugt, vorhält und zur Verfügung stellt. Eine solche doppelte Implementierung bedeutet aber immer zusätzlichen Aufwand bei der Erstellung und Pflege der Algorithmen, insbesondere besteht die Gefahr, dass Änderungen versehentlich nur an einer der beiden Implementierungen vorgenommen werden und so z.B. nur in der

Debug-Variante Fehler entfernt werden, diese aber in der später verwendeten Implementierung verbleiben.

Ein anderes interessantes Konzept stellt deshalb die „Debug-Shell“ dar. Es eignet sich zur Gewinnung von konkreten Informationen, die implizit in den Ein- und Ausgaben bzw. den Unterschieden zwischen diesen enthalten sind. Es handelt sich bei einer solchen „Debug-Shell“ um ein Objekt, dass die allgemeine Schnittstelle einer Art Filter implementiert, und jeden Filter, der dieselbe Schnittstelle implementiert, als „inneres“ Objekt aufnehmen kann. Die eingehenden Daten werden stets zunächst von der „Debug-Shell“ kopiert und dann unverändert an den beobachteten, den inneren Filter weitergeleitet. Dessen Ausgaben werden hinterher mit den ursprünglichen Eingangsdaten verglichen und so die entsprechenden Zusatzinformationen für den Benutzer gewonnen, bevor die Daten wiederum unverändert nach außen weitergeleitet werden.

Im dip-Framework wurde eine solche „Debug-Shell“ bereits für die Filter von Bildbewegungsdaten implementiert und zwar in der Klasse `cl_MovementFilterDebugShell`. Sie gibt aus, wie viele und welche Bildbewegungsdaten ein `i_MovementFilter`-Objekt in einem Schritt gelöscht hat und welche in diesem Schritt verblieben sind. (siehe auch Codebeispiel V.7)

In Zukunft soll dieses Konzept auch für andere Filtertypen z.B. für die Blob-Filter zum Einsatz kommen.

```
void dip::mdp::cl_MovementFilterDebugShell::filter()
{
   remaining.clear();
   erased.clear();
   std::vector<cl_MeasuredMovement>::iterator i;
   i=to_filter->begin();
   while (i<to_filter->end())
   {
      erased.push_back(*i);
      ++i;
   }
   inner_filter->filter();
   for(i=to_filter->begin();i<to_filter->end();++i)
   {
      erased.erase(std::find(erased.begin(),erased.end(),*i));
      remaining.push_back(*i);
   }
}
```

Codebeispiel V.5: Die Methode `filter()` *der „Debug-Shell“*

V.8. Kameraklassen und GUI

Die Kameraklassen des dip-Frameworks sind zur Zeit noch sehr rudimentär. Im Prinzip geben sie einfach nur ein Bild zurück. Dies soll später durch weitere Informationen wie Brennweite, Pixelgröße u.ä. ergänzt werden. Außerdem sollen sie Methoden zur Steuerung der Blende, der Fokussierung usw. erhalten. Alle Kameras implementieren die Schnittstelle `i_Camera` aus dem Modul `dip`. Diese wiederum ist von `i_UpdateObject` abgeleitet. Die Methode `update()` sorgt bei Kameras dafür, dass das Bild auf den neuesten Stand gebracht wird.

Zusätzlich zu den „realen" Kameras gibt es simulierte Kameras. Auch sie implementieren die Schnittstelle `i_Camera`. Simulierte Kameras bilden „echte" auf Basis von Filmen oder Einzelbildsequenzen[1] nach, wobei jeder simulierten Kamera genau ein Film zu Grunde liegt. Über ihre erweiterte eigene Schnittstelle (`i_SimCam`) kann man das Abspielen steuern. So lässt sich die Abspielgeschwindigkeit einstellen oder die aktuelle Position im Film abfragen und setzen. Das Abspielen kann gestartet und angehalten werden, Framedrops können simuliert werden. Zudem kann der Benutzer wählen, ob die Kamera beim letzten Bild eines Films stehen bleiben oder diesen von vorne wiederholen soll.

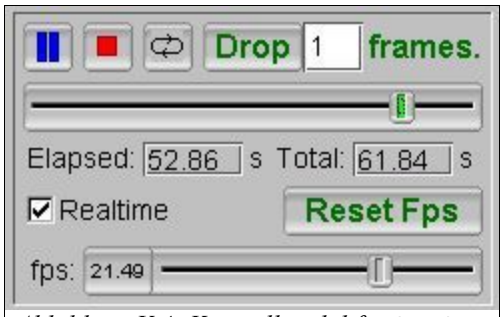


Abbildung V.4: Kontrollmodul für `i_SimCam`*-Objekte in der GUI (z.B. SPICE)*

Bildanalysealgorithmen bestehen häufig aus einer Kombination mehrerer Filter, die meist einige Parameter und Einstellmöglichkeiten besitzen, um sie an unterschiedliche Situationen und Aufgaben anpassen zu können. Für die Entwicklung neuer und das Testen bestehender Algorithmen unter diversen Bedingungen ist es daher nützlich, ein Werkzeug zur Verfügung zu haben, mit dem sich die Zwischenergebnisse und Ausgaben der Filter visualisieren lassen. Ebenso hilfreich ist es, wenn man die Parameterwerte der Filter, die Reihenfolge der Filter, ihre Anzahl und Art „online" ändern kann, während diese Filter die Bilder einer realen oder simulierten Kamera bearbeiten. So ist es möglich, direkt die Auswirkungen der Einstellungen zu beobachten und zu bewerten. Durch den Einsatz der simulierten Kameras, lassen sich wiederholt die verschiedenen Konfigurationen unter genau denselben Bedingungen testen und vergleichen.

1 Beides wird im Rest des Abschnitts der Einfachheit halber als „Film" bezeichnet.

Falls dann noch zusätzliche Informationen zu den Bildern vorliegen, wie beispielsweise Daten über die Position und Lage der Kamera, lassen sich noch weiterführende Erkenntnisse über die Güte der Algorithmen gewinnen und auch Vergleiche mit anderen Sensoren ziehen.

Daran anschließend brauchen nur noch die erfolgversprechenden Parametrisierungen in der Realität – also im Flugversuch getestet – werden. Die dabei gewonnen Daten, Bilder und Erkenntnisse können dann mit Hilfe dieses Werkzeugs aufgearbeitet und verfeinert werden.

Zusammen mit Olaf Guth wurde ein solches Werkzeug, SPICE (Smart Programm for Image Computing Experiments), entwickelt. Es handelt sich hierbei um eine grafische Benutzeroberfläche (GUI) für die Filter und Funktionen des dip-Frameworks. Neben den oben beschriebenen Funktionen gibt SPICE Auskunft über die aktuelle und durchschnittliche Bearbeitungszeit pro Bild. Man kann die Bearbeitungszeit künstlich verlangsamen und „Framedrops“ per Knopfdruck simulieren[1]. [3]

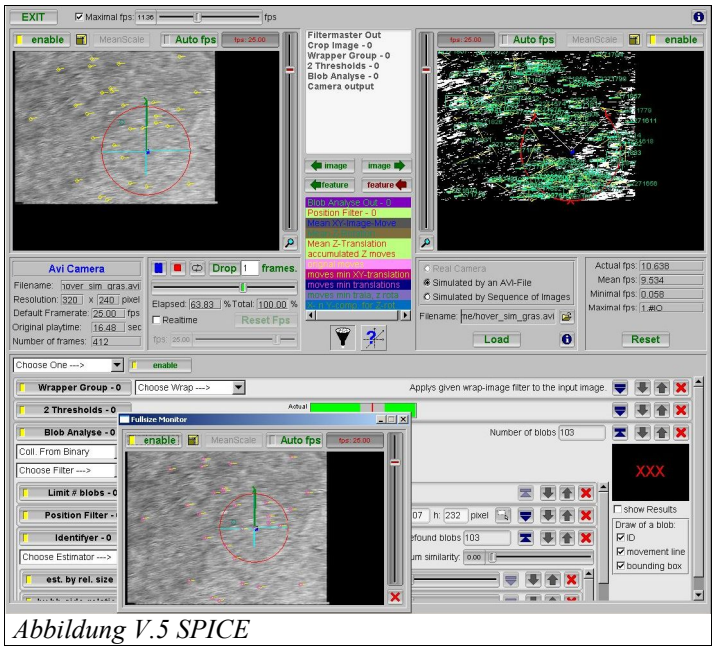


Abbildung V.5 SPICE

Beim Testen und Entwickeln von Bildverarbeitungsalgorithmen spielt diese GUI eine wichtige Rolle. Die Implementierung des Moduls `gui` ist aber für die in dieser Arbeit vorgestellten Verfahren eher nebensächlich. Daher wird auf die Erläuterung und Diskussion von Implementierungsdetails in dieser Arbeit verzichtet. Zum groben Überblick folgen nur ein paar Anmerkungen:

1 Dies ist nur mit simulierten Kameras möglich, s.o.

Das Modul `gui` wurde mit Hilfe der guten objektorientierten Bibliothek fltk (fast light toolkit) entwickelt. Es soll die Features und Einstellmöglichkeiten der Algorithmen und Filter möglichst vollständig nutzbar machen und deren Ausgaben möglichst deutlich klar und schnell erfassbar visualisieren. Gleichzeitig wurde darauf geachtet, dass die Implementierungen der Filter und Algorithmen vollkommen unabhängig von denen des Moduls `gui` bleiben. Prinzipiell ist das Modul `gui` daher so aufgebaut, dass es für jeden darzustellenden Filter bzw. jede darzustellende Klasse mindestens eine `gui`-Klasse gibt, die diese repräsentiert. Die Objekte dieser Klassen „beobachten“[1] den Zustand der von ihnen repäsentierten Objekte. D.h. sie fragen ihn regelmäßig ab, um ihre Anzeigen zu aktualisieren. Wenn die Anwendung schnell läuft, aktualisieren sich die (meisten) Anzeigen dabei nicht so häufig wie die Filter, sondern höchstens dreißig mal pro Sekunde, da der Mensch mehr auch nicht wahrnehmen könnte. Zusätzlich wurden Klassen zum Visualisieren z.B. von Drehwinkeln oder Bewegungsvektoren und zur komfortablen Eingabe[2] erstellt. [15]

V.9. Beispielaufbau und Test der Blob-Analyse-Algorithmen

Nachdem in den vorherigen Kapiteln und Abschnitten die einzelnen Algorithmen und Konzepte zur Schätzung der Eigenbewegung mit Hilfe der Blob-Analyse sowie zuletzt das geschaffene Bildverarbeitungs-Framework vorgestellt wurden, soll in diesem Abschnitt das Zusammenspiel dieser Komponenten in der Praxis an einem Beispielaufbau erläutert werden (Abbildung V.6 auf Seite 79 zeigt den Beispielaufbau und die Parametrisierung der Filter in der GUI).

Generell ist die Komplexität und Ausführungszeit bei Filter-Algorithmen abhängig von der Anzahl der zu filternden Objekte. Es ist daher eine gute Strategie, die Anzahl der betrachteten Objekte frühzeitig zu verringern. Ebenso sollte man die Komplexität der Algorithmen bei deren Wahl und Anordnung beachten. Hochkomplexe Algorithmen sollten, soweit möglich, erst zum Einsatz kommen, wenn die Zahl der zu filternden Objekte sehr klein geworden ist. Am Anfang hingegen ist eher auf Algorithmen mit kleiner, z.B. linearer Komplexität zurückzugreifen. Komplexität und Objektanzahl verhalten sich im Beispiel wie in der folgenden Tabelle zu sehen.

---

1 Meint nicht eine Implementierung nach dem Beobachter-Entwurfsmuster. Dort müssten die Bildverarbeitungsalgorithmen stets ein „notify()“ für ihre Beobachter aufrufen, was unnötig Zeit kosten kann.

2 Z.B. wird mit der Maus durch „Aufziehen“ eines Rahmens ein Bildausschnitt ausgewählt.

Verwendete (Filter-) Klasse	Anzahl Eingangs-Objekte (n)	Komplexität des Algorithmus $O(n)$
`cl_CroppedImage`	307200 Pixel	$<n$
`cl_2ThreshFilter`	153600 Pixel	n
`cl_BlobCollectFromBinary`	153600 Pixel	n
`cl_PositionFilter`	~1500-5800 Blobs	n
`cl_LimitNumBlobs`	~1300-5500 Blobs	$n * \log(n)$
`cl_Identifyer`	75 Blobs	n^2
`cl_SmoothLocalMoves`	~47-62 Bildbewegungen	$>n^2$
`cl_SimpleMovesEstimator`	~15-25 Bildbewegungen	n

Da das von der Kamera gelieferte Bild mit 640 x 480 Pixeln recht groß ist, größer als in der Regel zum Erhalt von genügend vielen ausreichend großen Blobs zum Messen der Eigenbewegung nötig wäre, wird zunächst ein kleinerer Ausschnitt von 480 x 320 Pixeln gewählt (siehe auch Abschnitt V.3.3 „Maskierte Bilder“).

Die in diesem Ausschnitt vorhandenen Bilddaten werden in ein Binärbild umgewandelt. Besonders helle oder dunkle Pixel im Bild werden dabei weiß markiert. Die Grenzwerte, ab denen Pixel als „besonders hell“ bzw. „besonders dunkel“ gelten, werden dynamisch – relativ zur Durchschnittshelligkeit des zuvor betrachteten Bildes – festgelegt. Das hat den Vorteil, dass der Algorithmus auf Änderungen der Lichtverhältnisse u.ä. angepasst reagieren kann. Aus dem Binärbild werden dann mit einem optimierten Algorithmus alle weißen Flecken als „Blobs“ extrahiert (siehe auch Abschnitt III.4.2 „Blobs aus Binärbildern“).

Da Blobs, die den Rand berühren, nicht vollständig (zu sehen) sein können, werden sie heraus gefiltert, um Fehlidentifizierungen zu vermeiden.

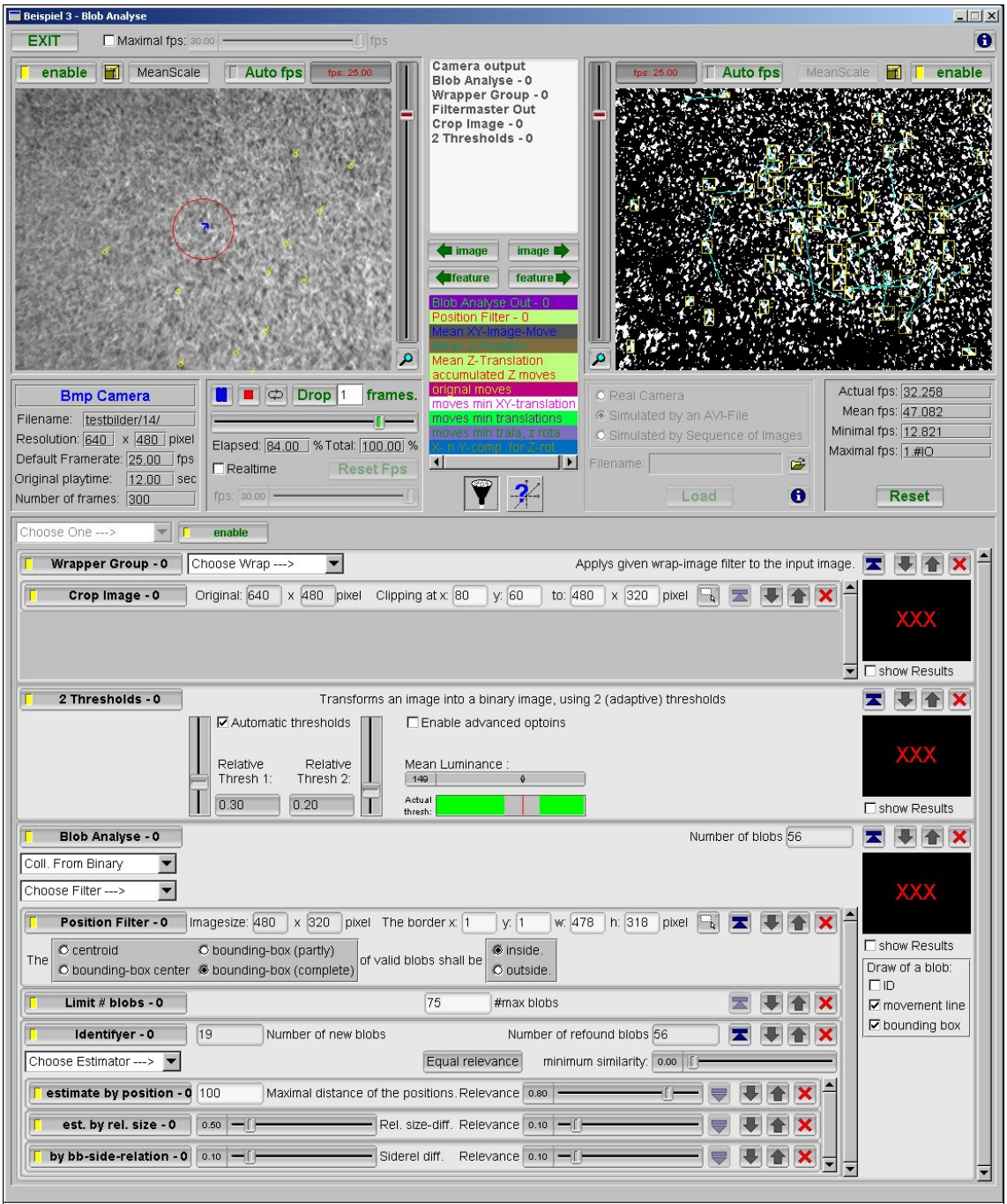


Abbildung V.6: Beispielaufbau mit GUI zur Eigenbewegungsschätzung mit Hilfe der Blob-Analyse.

Danach wird die Anzahl der Blobs auf 75 reduziert. Das sind ausreichend viele, so dass nach Identifizierung und Bewegungsfilterung noch genügend Daten zur Bestimmung der Eigenbewegung übrig bleiben. Es werden jeweils die 75 größten Blobs ausgewählt, da große Blobs einfacher und sicherer zu identifizieren sind als kleine (siehe auch Abschnitt III.5 „Filter für Blobs“).

Nun werden die Blobs identifiziert und ihre Bewegung im Bild ermittelt. Neben Größe und grober Form wird hierbei besonders die Position zur Bestimmung der Ähnlichkeit herangezogen, da davon auszugehen ist, dass alte und neue Position eines Blobs jeweils benachbart sind (siehe auch Abschnitt III.6 „Blob-Identifizierung“).

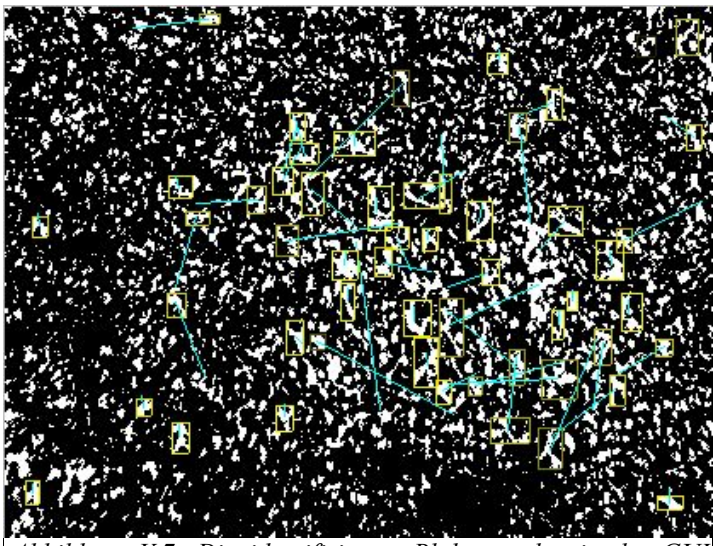


Abbildung V.7: Die identifizierten Blobs werden in der GUI zum Beispielaufbau durch ihre „bounding-box“ und ihre Bewegungslinien zusammen mit dem Binärbild visualisiert.

Die ermittelten Bewegungsdaten werden an die Algorithmen zur Schätzung der Eigenbewegung weitergegeben (siehe auch Kapitel IV „Eigenbewegung“).

Mit Hilfe der Klasse `cl_SmoothLocalMoves` werden nun zunächst die Bewegungsdaten gefiltert. Nach Blob-Identifizierung und Bildbewegungsdaten-Filterung bleiben dann ca. 15-25 Daten über. Damit wird anschließend die relative Eigenbewegung geschätzt.

Die Ergebnisse dieser Schätzung werden für den Benutzer visualisiert. (siehe Abbildung V.8) Es ist dabei zu erkennen, dass dieser Aufbau die Eigenbewegung schon sehr gut schätzt. Einige Fehlmessungen lassen sich darauf zurückführen, dass die Blob-Identifizierung bisher ein wenig ungenau ist, da sie auf zu wenigen Merkmalen basiert. Andere Abweichungen ergeben sich aus dem nur zu Test- und Visualisierungszwecken implementierten Algorithmus zur Schätzung der Eigenbewegung. Die Algorithmen, die derzeit neu implementiert werden, dürften hier Abhilfe schaffen. Außerdem fällt auf, dass bereits dieser Aufbau trotz Visualisierung und GUI ungefähr in Echtzeit (je nach eingesetztem Testrechner auch um einiges schneller) arbeitet.

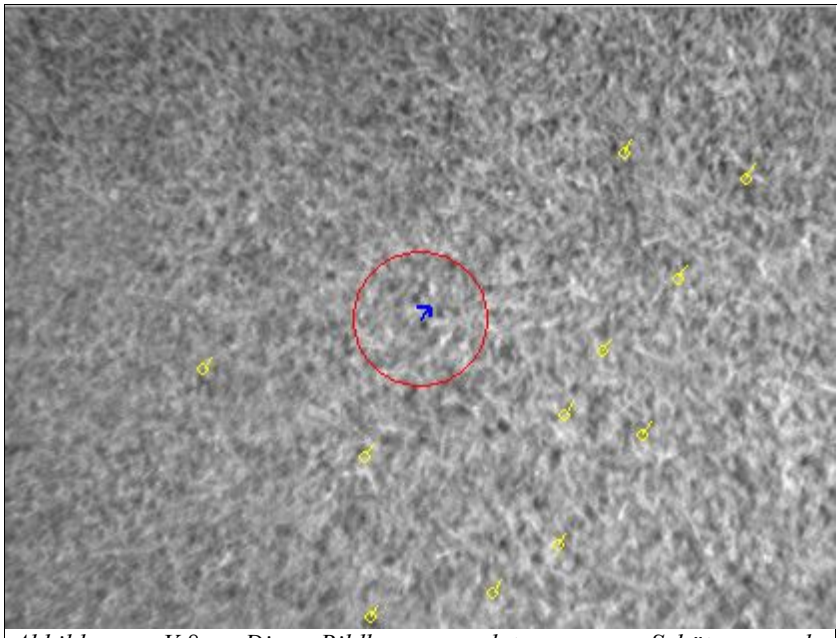


Abbildung V.8: Die Bildbewegungsdaten zur Schätzung der Eigenbewegung werden im linken Monitor der GUI zum Beispielaufbau zusammen mit dem original Kamerabild angezeigt. Die geschätzte Translation parallel zur Bildebene wird als blauer Pfeil dargestellt. Zur Visualisierung der Translation senkrecht zur Bildebene ist der rote Kreis gedacht: Sinkt der Hubschrauber so wird der Radius des Kreises größer; steigt der Hubschrauber, so wird der Radius kleiner.

Zum Vergleich wurde auf denselben Bildausschnitt der bisher häufig eingesetzte um Bildpyramiden erweiterten Lucas-Kanade-Feature-Tracker angewandt[1]. Die benötigten Features werden hierbei mit Hilfe der Funktion „`cvGoodFeaturesToTrack()`“ aus der Bildverarbeitungsbibliothek OpenCV [13][20] zur Verfügung gestellt. Auf den Filter zur Filterung der Bildbewegungsdaten wurde verzichtet, da der Feature-Tracker nicht so starke Ausreißer erzeugt wie (zur Zeit noch) die Blob-Analyse. Die Anzahl der zu betrachtenden Features wurde auf den Bereich [15,25] festgelegt, was der Anzahl der effektiv zur Schätzung der Eigenbewegung genutzten Daten bei der Blob-Analyse entspricht.

Vergleicht man die Ausgaben der Schätzung der Eigenbewegung mit Hilfe der Blob-Analyse mit den mittels Feature-Tracker geschätzten Ergebnissen in der GUI, so sind sie in der Regel sehr ähnlich. Meist erzielt der Feature-Tracker etwas bessere Ergebnisse als die Blob-Analyse. Bei großen Bildbewegungen liegt aber die Blob-Analyse im Vorteil, da beim Feature-Tracker die betrachteten Features oft „durchrutschen“, d.h. es wird eine geringere Bewegung festgestellt,

1 Es wurde die Implementierung des Feature-Trackers innerhalb des dip-Frameworks von Olaf Guth verwendet. [3][4]

als tatsächlich vorhanden. Probleme hat die Blob-Analyse zur Zeit mit sehr feinen oder gleichförmigen Strukturen. So sind z.B. einige der Labor-Aufnahmen über einem fein-strukturierten, kontrastarmen Teppich für die Blob-Analyse kaum zu verwenden, da dort zu wenig charakteristische Blobs gefunden werden.

Über natürlichem Untergrund wie, z.B. Gras sind solche Probleme kaum festzustellen. Störend in Flugversuchen wirkt sich höchsten die Abgasfahne des Helicopters aus, falls diese größere Teile des Bildes überdeckt. Eine erste Maßnahme zur Abhilfe kann hier ein Binärbild sein, dass nur die besonders dunklen, nicht die besonders hellen Pixel markiert, da die Abgasfahne sehr hell ist.

Ein Ziel bei der Wahl der Blob-Analyse als Bildverarbeitungsverfahren war es, einen Algorithmus mit hoher Ausführungsgeschwindigkeit zu finden. Zu einem ersten Vergleich der Ausführungsgeschwindigkeit der beiden Algorithmen wurde der Beispielaufbau jeweils für die Blob-Analyse und entsprechend für den Feature-Tracker mit denselben Einstellungen wie oben beschrieben, aber ohne GUI und Ausgaben zur Laufzeit der Algorithmen, erstellt. Jeweils eine Bilderserie mit 300 Bildern wird von diesen Applikationen abgearbeitet und danach die durchschnittliche pro Bild benötigte Bearbeitungszeit ausgeben. Hier zeigt sich, dass die Blob-Analyse um einiges schneller arbeitet als der verwendete Feature-Tracker (auf einem Testsystem mit einem Athlon 2500+ Prozessor und 512 MB RAM erreichte der Blob-Analyse-Aufbau eine mittlere Framerate von 59.5 fps, der Aufbau mit dem Feature-Tracker brachte es im Durchschnitt auf eine Framerate von 36.6 fps).

V.10. Ausblick auf die weitere Entwicklung des Frameworks

Obwohl es schon so viele Klassen und Funktionalitäten im dip-Framework gibt, ist es doch weit davon entfernt, vollständig oder gar endgültig fertig zu sein. Es gibt Einiges, was noch verbessert und Vieles, was noch ergänzt werden sollte.

Neben weiteren Filtern und Bildverarbeitungsalgorithmen gehört dazu auch ein plattformunabhängiges Modul zum Zugriff, Erstellen und Modifizieren von Dateien. Mit dessen Hilfe könnte man dann ein eigenes Dateiformat realisieren, in dem man die Konfiguration und Anordnung von Filtern z.B. in SPICE laden und speichern könnte und das auch dazu benutzt werden kann, solche Konfigurationen in andere Testprogramme ohne GUI z.B. zum Test mit ARTIS im Feld zu laden. Des Weiteren könnte man zu Filmen und Bildsequenzen kameraspezi-

fische Informationen wie Brennweite, Pixelabstand usw. auf der Festplatte ablegen und diese dann bei der Simulation zur Verfügung stellen. Ebenso wäre es denkbar, auf die aufgezeichneten Daten des Navigationscomputer zurück zu greifen, um die Ergebnisse der einzelnen Algorithmen zu bewerten[1].

Auch ein Eventsystem könnte, zumindest für Simulationen und GUI-Anwendungen, hilfreich sein, z.B. um mitzuteilen, dass eine simulierte Kamera das Abspielen eines Films beendet hat und von vorn beginnt. In diesem Fall könnten Filter usw. beim Eintreten des Events neu initialisiert werden.

Darüber hinaus sind auch weitere simulierte Kameras denkbar. Z.B. eine, die in einer per OpenGL generierten Welt (mit Fototexturen als Untergrund), von den Bildverarbeitungsalgorithmen angesteuert, bewegt wird. Es könnte auch das Bild von virtuellen Kameras aus den zum Testen der Steuerung verwendeten Flugsimulatoren übertragen werden. Hierbei könnten dann die Ausgaben der Bildverarbeitung direkt an die Navigationsalgorithmen übergeben werden, um so das Zusammenspiel von Bildverarbeitung und Navigation zu testen.

Filter, die es ermöglichen, die Bearbeitung der Bilder in verschiedene quasi parallel laufende Algorithmenstränge zu übertragen, sind notwendig, um z.B. mehrere Aufgaben gleichzeitig zu bearbeiten. Wenn man solche Stränge dann wieder zusammenführen kann, lassen sich zusätzliche Synergieeffekte nutzen und vielfältige andere, neue Algorithmen implementieren.

Auch andere weitergehende Visualisierungsmöglichkeiten können sinnvoll sein, z.B. eine dreidimensionale Ansicht des Hubschraubers relativ zu einer Bodenfläche, um die gemessene Lage und Bewegung einfach vorstellbar zu machen. Oder die eingefärbte Überlagerung mehrerer Ergebnisbildern in den Monitoren.

Wie erwähnt soll das Konzept der „Debug Shells“[2] noch für weitere Filterklassen implementiert werden. Selbst diese umfangreiche Auflistung gibt nur einen Ausschnitt der vorhandenen Ideen wieder.

1 Hierzu wären wiederum entsprechende Auswertungsklassen hilfreich.

2 Näher erläutert in Abschnitt V.7.

VI. Fazit und Ausblick

VI.1. Fazit: Eignung der Blob-Analyse zum relativen Hovern

Als Alternative zu den vorhandenen Algorithmen zum relativen Hovern wurde die Blob-Analyse untersucht. Sie sollte Bildbewegungsdaten möglichst schnell ermitteln, welche dann zur Schätzung der Eigenbewegung mit Hilfe bekannter Verfahren eingesetzt werden können. Die Blob-Analyse ist sehr gut mit Hilfe des Konzepts der Objektorientierung implementierbar und begünstigt damit die Entwicklung hochdynamischer Anwendungen, die sich gut verschiedenen veränderbaren Bedingungen anpassen können.

Die im Rahmen dieser Arbeit implementierten Algorithmen zur Blob-Analyse stellen dabei einen Ansatz, einen Anfang dar. Mit ihrer Hilfe wurde versucht auszuloten, ob und wie sich die Blob-Analyse zum relativen Hovern, bzw. zur Gewinnung von Bildbewegungsdaten zur Bestimmung der Eigenbewegung eignet. Tests im erstellten dip-Framework zeigten, dass die Blob-Analyse sich insbesondere zum Einsatz über heterogenem, natürlichem Untergrund wie z.B. Gras eignet. Die meisten Eigenbewegungen werden richtig erkannt, und die vorgestellten Algorithmen erweisen sich als sehr schnell.

Die Blob-Analyse eignet sich aufgrund ihrer hohen Geschwindigkeit insbesondere zur zeitkritischen Online-Bildverarbeitung. Sie kann damit also zum relativen Hovern oder aber auch zur unterstützenden Messung der Eigenbewegung – z.B. falls andere Sensoren ausgefallen sind oder um deren Ausgaben zu validieren – eingesetzt werden. Hierzu sollte sie noch weiter entwickelt werden. Verglichen mit dem Feature-Tracker ist die Blob-Analyse weniger rechenintensiv, aber etwas ungenauer. Eine Schätzung der Eigenbewegung bei der die Bildbewegungsdaten mit Hilfe der Blob-Analyse ermittelt werden, liefert Ergebnisse, die in der Genauigkeit vergleichbar sind zu denen einer Schätzung mit Hilfe des Feature-Trackers. Die Ergebnisse der Blob-Analyse sind lediglich etwas verrauschter. Bei grossen Bildbewegungen ist die Blob-Analyse dem Feature-Tracker aber überlegen. Da im Moment noch nicht alle Möglichkeiten zum Wiederfinden von Blobs voll ausgeschöpft werden[1], ist also noch einiges Potential zur Verbesserung der Qualität der Blob-Analyse vorhanden.

1 z.B. noch sehr wenige Blobmerkmale betrachtet werden, bessere Filter möglich sind (siehe auch Abschnitt III.7)

Das erstellte dip-Framework ermöglicht die einfache Implementierung und das schnelle Testen von Bildverarbeitungsalgorithmen. Es ist gut erweiterbar und bietet vielfältige Möglichkeiten der Visualisierung der Parameter und Ergebnisse. Mit Hilfe des neu entwickelten Tools SPICE können Parametrisierung und Anordnung von verschiedenen Bildverarbeitungsalgorithmen online getestet und verändert werden. Der Benutzer erhält dabei ein direktes Feedback über die Auswirkungen der gemachten Einstellungen. Das ist sehr hilfreich insbesondere bei der Einwicklung neuer Filter und Verfahren, da nicht erst ganze Durchläufe oder das Auswerten von Log-Dateien abzuwarten sind, um auf die Ergebnisse und das Verhalten bestimmter Konfigurationen zu reagieren. Die bereitgestellten Bildklassen und Schnittstellen schaffen die Voraussetzungen für eine schnelle und effiziente Verarbeitung der Bilddaten. Durch die strikte Unabhängigkeit der Bildverarbeitungsalgorithmen von der GUI und den Visualisierungsklassen können einfach funktionierende Konfigurationen als schnelle Applikation ohne Zusatzausgaben für den Benutzer erstellt werden. Diese können dann direkt in der Praxis eingesetzt werden. Das dip-Framework kann somit als Basis für weitere Arbeit und Forschung im Bereich der Blob-Analyse sowie der gesamten digitalen Bildverarbeitung unter anderem im ARTIS Projekt dienen.

VI.2. Ausblick

Aufgrund ihrer unterschiedlichen Vorzüge und Nachteile, wäre es eine sinnvolle Möglichkeit den Feature-Tracker und die Blob-Analyse gemeinsam bzw. sich gegenseitig ergänzend in einer Anwendung zu gebrauchen. So wäre es möglich, falls die Ergebnisse der Blob-Analyse zu ungenau werden, bzw. wenn sie zu wenig Daten zur sicheren Schätzung der Eigenbewegung erzeugt (weil beispielsweise Abgase das Bild stören oder der Untergrund sehr feine Strukturen aufweist), auf den langsameren Feature-Tracker umzuschalten, und umgekehrt die Blob-Analyse zu bevorzugen, falls die Bildbewegungen so groß werden, dass sie vom Feature-Tracker nicht sicher genug gemessen werden können (weil beispielsweise der Hubschrauber dicht über dem Boden fliegt). Das erstellte dip-Framework würde ein solches Umschalten, wie auch den gleichzeitigen Betrieb beider Verfahren gut unterstützen, da zum einen die Algorithmen durch die gemeinsame Filter-Schnittstelle einfach austauschbar sind, und zum anderen durch das allgemeine, von den Algorithmen unabhängige Format die Bildbewegungsdaten jeweils direkt von den Algorithmen zur Schätzung der Eigenbewegung ausgewertet werden können.

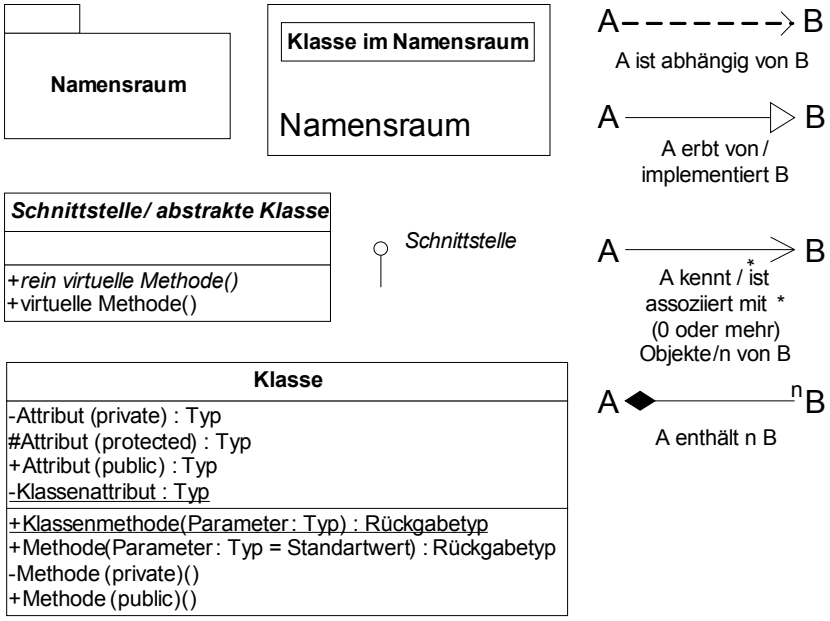
In naher Zukunft sollten die vorhandenen Ansätze und Implementierungen zur Blob-Analyse ausgebaut werden, so dass diese genauer und mit weniger Messfehlern die Blobs und ihre Bewegungen identifizieren kann, ihr Potential besser ausschöpft. Es sollten Filter und Algorithmen rund um die Blob-Analyse erstellt werden, die mit ihr Zusammenarbeiten und sie noch universeller und effektiver einsetzbar machen. Die Parameter, die zur Zeit noch vom Benutzer „von Hand" gesetzt werden, können automatisiert und auf Gegebenheiten im Bild dynamisch angepasst werden. Ist z.B. die Menge der gefundenen Blobs zu groß oder zu klein so könnte der betrachtete Bildausschnitt innerhalb gewisser Grenzen erweitert, bzw. verkleinert werden. Grenzwerte für die Identifizierung könnten im Laufe der Zeit von den Algorithmen an die Art der vorgefundenen Blobs angepasst werden usw.

Mit Hilfe des erstellten Frameworks und des Testtools SPICE werden viele Algorithmen der Bildverarbeitung im ARTIS Projekt entwickelt und getestet werden können. Besonders lohnende, gut funktionierende Algorithmen, die zur Zeit noch mit Hilfe von Fremdbibliotheken implementiert sind, werden noch effizienter direkt im Framework realisiert. Zusätzlich ist es wünschenswert, dass Konstrukte dem Framework hinzugefügt werden, so dass eine parallele Bearbeitung mehrerer Aufgaben gleichzeitig möglich wird. Dann kann der Vorteil der Bildverarbeitung als flexibles „Multifunktionswerkzeug" richtig genutzt werden.

Dabei wird gerade die Flexibilität der objektorientierten Software ein entscheidender Aspekt sein. Früher wurde noch sehr häufig darüber nachgedacht, Bildverarbeitungsalgorithmen direkt „in Hardware" zu realisieren. Doch inzwischen ändert sich die Entwicklungsrichtung zunehmend. Denn einerseits werden billige Standardkomponenten, wie PC-Hardware immer leistungsfähiger und schneller (und zwar in äußerst kurzen Zeitabschnitten), andererseits wird die Bildverarbeitung immer komplexere Aufgaben mit dynamischen Anforderungen und Zielsetzungen zu erledigen haben. Dies erfordert eine hohe Flexibilität, die mit spezialisierter Hardware nicht zu erreichen ist. Die Software (wie z.B. das dip-Framework mit seiner hohen Plattformunabhängigkeit) hat zusätzlich den Vorteil, dass sie einfach auf neue, bessere Standard-Hardware übertragen werden kann. So werden objektorientierte Bildverarbeitungsalgorithmen und Bildverarbeitungssoftware in den nächsten Jahren immer mehr an Bedeutung gewinnen, und im ARTIS Projekt nicht nur zum relativen Hovern eingesetzt werden.

VII. Anhänge

VII.1. Verwendete Symbole in UML-Diagrammen



Anmerkung: Die meisten UML-Diagramme in dieser Arbeit zeigen die Klassen, ihre Attribute und Methoden der Übersicht halber nicht vollständig. Es wurden die Teile weggelassen, welche im jeweiligen Zusammenhang nicht wichtig erschienen.

VII.2. Farbkonventionen in Codebeispielen

Farbe	Codeelemente	Beispiele
blau	C++-Schlüsselwörter, definierte und eingebaute Typen, Namensräume, Aufzählungen, Strukturen, Klassen, Schnittstellen	`new`, `double`, `typedef int 32bit_integer`, `dip::filter`, `e_BlobFilterType`, `st_Point`, `cl_Blob`, `i_Image`
rotbraun	Funktionen, Methoden	`main()`, `the_blob->getCentroid();`
grau	Konstanten, Variablen	`c_PI`, `the_blob`
violett	Aufzählungselemente, Präprozessormakros	`E_2THRESHOLDS`, `MATH_CONSTANTS_H`
grün	Kommentare	`// Forward declaration`

VII.3. Präfixbedeutungen

Präfix	Codeelement	Beispiel
i_	Schnittstelle	i_Filter
cl_	Klasse	cl_Image
st_	Struktur	st_Point
e_	Aufzählung	e_FilterType
c_	Konstante	c_PI
E_	Aufzählungs-element	E_TRACKER

VII.4. Abbildungsverzeichnis

VII.5. Verzeichnis der UML-Diagramme

VII.6. Verzeichnis der Codebeispiele

VII.7. Literaturverzeichnis

[1] J. S. Dittrich, A. Bernatz, and F. Thielecke: „**Intelligent systems research using a small autonomousrotorcraft testbed**“, 2nd AIAA "Unmanned Unlimited" Conf., Workshop and Exhibit, Nr. 2003-6561 in AIAA Papers, (San Diego, CA), September 2003

[2] A. Bernatz und F. Thielecke: „**Bildgestützte Schwebeflugstabilisierung für einen autonomen Hubschrauber**“ , Deutscher Luft- und Raumfahrtkongress, Jahrbuch 2004

[3] Olaf Guth: „**Biologisch inspirierte Bildverarbeitungsalgorithmen zur Realisierung eines Geländefolgefluges für einen autonomen Kleinhubschrauber**“, Fachbereich Informatik in der Fachhochschule Braunschweig/Wolfenbüttel, 2004

[4] J.-Y. Bouguet: „**Pyramidal Implementation of the Lucas Kanade Feature Tracker, Desrciption of the Algorithm**“, Intel Corporation, Microprocessor Research Labs

[5] C. Tomasi and T.Kanade: „**Detection and Tracking of Point features**“, Technical Report CMUCS-91-132, Carnegie Mellon University, 1991

[6] C. S. Sharp, O. Shakernia, and S. Sastry: „**A vision system for landing an unmanned aerial vehicle**“, in Proceedings of IEEE ICRA, (Seoul, Korea),S. 1720 { 1727, IEEE ICRA, Mai 2001

[7] O. Masoud and N. P. Papanikolopoulos: „**A Robust Realtime Multilevel Modelbased Pedestrian Tracking System**“, Artificial Intelligence, Robotics, and Vision LaboratoryDepartment of Computer Science,University of Minnesota, 1997

[8] J. Renno, J. Orwell and G.A. Jones: „**Learning Surveillance Tracking Models for theSelf-Calibrated Ground Plane**“, Digital Imaging Research Centre, Kingston University, 2002

[9] N. M. Josuttis : „**The C++ Standard Library: A Tutorial and Reference**“, 1. August 1999, Addison-Wesley Professional, ISBN: 0201379260

[10] H. Adams, S. Singh and D. Strelow: „**An empirical comparison of methods for image-based motion estimation**“, IEEE/RSJ International Conference on Intelligent Robots and Systems, October 2002

[11] W. Chojnacki, M. J. Brooks, A. van den Hengel, D. Gawley : „**Revisiting Hartley's normalised eight-point algorithm** “, IEEE Trans. Pattern Analysis Machine Intelligence, 25, 9, 2003, 1172-1177

[12] Collins Cobuild : „**English Dictonary**“, 1995, p. 166 „blob“, HarperCollins Publishers, ISBN: 0003750299

[13] Intel Corporation, „**Open Source Computer Vision Library**“, `http://www.intel.com/research/mrl/research/opencv/`

[14] Chair of Technical Computer Science, RWTH Aachen University, „**LTI-Lib Computer Vision Libary**“, `http://ltilib.sourceforge.net/doc/homepage/index.shtml`

[15] B. Spitzak and others, „**FLTK, The Fast Light Toolkit**“, `http://www.fltk.org/`

[16] E. Gamma, R. Helm, R. Johnson, J. Vlissides : „**Entwurfsmuster - Elemente wiederverwendbarer objektorientierter Software**“, Juli 2004, Addison-Wesley, München, ISBN: 3827321999

[17] E. Gabrilovich: „**Controlling the Destruction Order of Singleton Objects**“, C/C++ Users Journal, October 1999

[18] John Vlissides : „**Pattern Hatching: Design Patterns Applied**“, 1. Juni 1998, Addison-Wesley Professional, ISBN: 0201432935

[19] Scott Meyers : „**Effektiv C++ programmieren . 50 Möglichkeiten zur Verbesserung Ihrer Programme**“, 3. Aufl. 15. Dezember 1997, Addison-Wesley, ISBN: 3827313058

[20] J. Shi and C. Tomasi: „**Good Features to Track.**“, Proc. IEEEConf. Computer Vision and Pattern Recognition, pp. 593-600, 1994

[21] Y. Ma: „**Differential Geometric Approach to Computer Vision and its application in control**“, Dissertation f. PhD, University of California at Berkeley, 2000

[22] A documentation system for C++, C, Java and others, „**Doxygen**“, http://www.doxygen.org