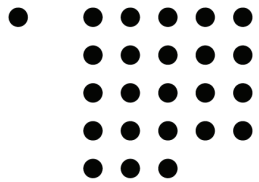

Fachhochschule Köln
University of Applied Sciences Cologne
Campus Gummersbach

Fachbereich Informatik
Studiengang Allgemeine Informatik

Diplomarbeit

Automatische und videobasierte Erstellung einer Endoskopmaske

Vitaly Morozov
Matrikelnummer: 11028442

Erstprüfer: Prof. Dr. Wolfgang Konen
Zweitprüfer: Prof. Dr. Gerhard Plaßmann

31 August 2009

Zusammenfassung

Im Rahmen dieser Arbeit wurde ein bestehendes Programm zum Image Mosaicing für die medizinische Endoskopie, d.h. zur Konstruktion eines Übersichtsbildes aus einer Folge von endoskopischen Einzelaufnahmen, weiterentwickelt. Eine erweiterte Benutzerschnittstelle und die erweiterten Möglichkeiten der automatisch erstellten Maske erlauben einen flexiblen Zugriff auf gespeicherte Aufnahmen.

Die Implementierung erfolgte betriebssystemunabhängig in Java unter Verwendung der Programmbibliothek ImageJ (ij.jar). Die erstellten Programmerweiterungen erlauben die systematische Evaluierung des Verfahrens auf vorhandenen Datensätzen und stellen einen wesentlichen Schritt in Richtung einer klinischen Erprobung dar.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Durch Image Mosaicing[1][4][5] ist es möglich aus einer Videosequenz oder einzelnen Bildern einer Bildfolge einen Übersichtsbild (sogenanntes Panoramablick) zu erstellen. In der Medizin und speziell in der Endoskopie könnte so ein Übersichtsbild dem operierenden Arzt eine bessere Orientierung verschaffen und ihm somit die Arbeit erleichtern.

Ziel dieser Arbeit ist die Erweiterung des bestehenden Softwareprojektes für Image Mosaicing. Es wird die Möglichkeit angestrebt, die Maske für Nutzbereich des endoskopischen Bildmaterials automatisch zu erstellen, und somit die medizinische Evaluierung der Software zu ermöglichen.

1.1 Motivation

Das hier vorgestellte PlugIn, das auf von Kourogi [1] vorgestellten Algorithmus basiert [2],[3], stellt eine solide Vorstufe zur dem Einsatz der Software bei endoskopischen Operationen dar.

Eins der Hindernisse, die auf dem Weg der medizinischen Evaluierung stehen, beruht auf der Tatsache, dass die Masken für den Nutzbereich im Moment noch manuell erstellt werden müssen. Eine automatische Maskenerstellung wurde einen weiteren Schritt hin zur Einführung darstellen und die Massentest ermöglichen.

1.2 Zielsetzung

Wie bereits erwähnt, ist das Ziel dieser Arbeit - die automatische Erstellung der Maske für den

Nutzbereich des endoskopischen Bildmaterials. Dazu gehörten die Untersuchungen mehrerer endoskopischer Aufnahmen und die Suche nach passenden Algorithmen.

Am Ende dieser Arbeit soll es möglich sein, Image Mosaicing ohne vorläufige manuelle Maskenerstellung zu starten. Die Maske soll möglichst genau sein (Abbildung 1.1). Das heißt die Maske soll möglichst den ganzen Nutzbereich erschließen (Abbildung 1.2) und vor allem möglichst keine Hintergrundanteile im Nutzbereich aufweisen (Abbildung 1.3).

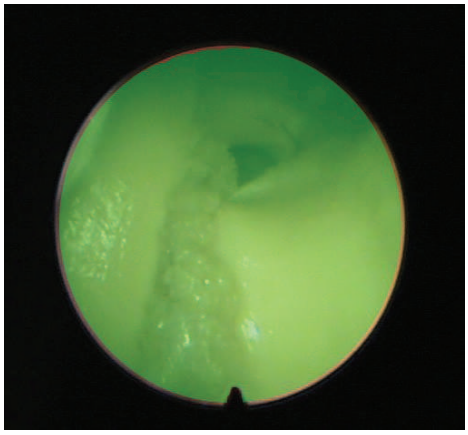

Abbildung 1.1: Optimale Maske

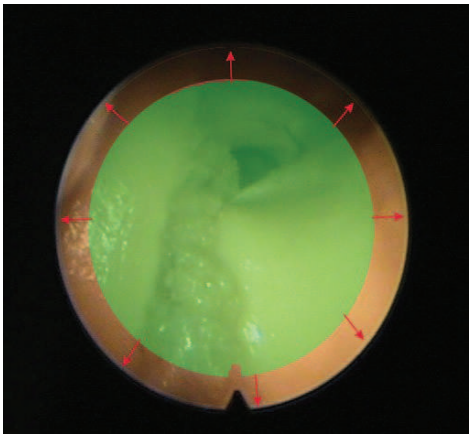

Abbildung 1.2: Nicht optimale Maske

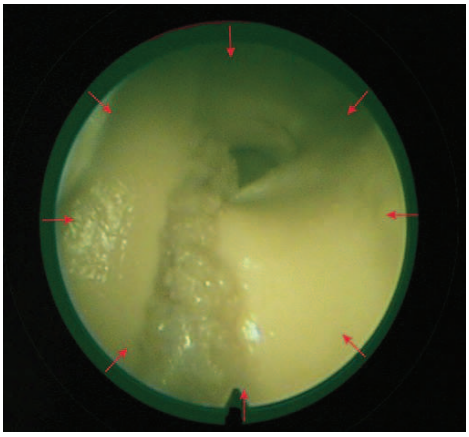

Abbildung 1.3: Hintergrundanteile im Nutzbereich

1.3 Aufbau der Arbeit

Um dem Leser vorab einen Überblick zu verschaffen, soll im Folgenden kurz auf die Inhalte der einzelnen Kapitel dieser Arbeit eingegangen werden.

Zunächst werden im Kapitel 2 die allgemeinen Grundlagen angesprochen, die für das Verständnis der Arbeit wichtig sind. Diese beinhalten mathematische Grundlagen der Algorithmen, die in der Arbeit benutzt werden.

Im Kapitel 3 wird die technische Realisierung angegangen. Es wird zuerst die Grundidee von der Maskenerstellung erklärt. Daraufhin werden die Listings mit Quellcode vorgestellt, und ausführlich erklärt. Das Kapitel schlisst mit der Erklärung der graphischen Benutzeroberfläche.

Im Kapitel 4 werden Resultate der Test vorgestellt sowie die optimale Einstellungen der Benutzeroberfläche besprochen.

Im Kapitel 5 folgt das Fazit.

Kapitel 2

Grundlagen

Für das Verständnis des Themengebietes bedarf es einiger einführender Erläuterungen. Auf diese soll in dem folgenden Kapitel näher eingegangen werden.

2.1 Canny-Kantendetektion

Der Canny-Algorithmus [6][8], benannt nach John Canny, ist ein in der digitalen Bildverarbeitung weit verbreiteter, robuster Algorithmus zur Kantendetektion. Er gliedert sich in verschiedene Faltungsoperationen und liefert ein Bild, welches idealerweise nur noch die Kanten des Ausgangsbildes enthält. Eine Kante wird dabei als Änderung der Grauwerte betrachtet. Je stärker die Änderung, desto höher die Kantenstärke. Auch die Kantenrichtung ist interessant. Abbildung 2.1 zeigt eine starke Kante in horizontaler Richtung und eine schwache Kante in vertikaler Richtung.

Abbildung 2.1: Kanten

Der Algorithmus kann nur Graubildern bearbeiten, deswegen ist eine vorherige Überführung von farbigen Bildern in Graubilder erforderlich.

Der Algorithmus arbeitet in 4 Schritten:

Schritt 1.

Da große Helligkeitsschwankungen auch ohne das Vorhandensein von Kanten einfach durch Bildrauschen auftreten können, ist eine sogenannte Normalverteilung (Abbildung 2.2) zur Glättung des Bildes notwendig.

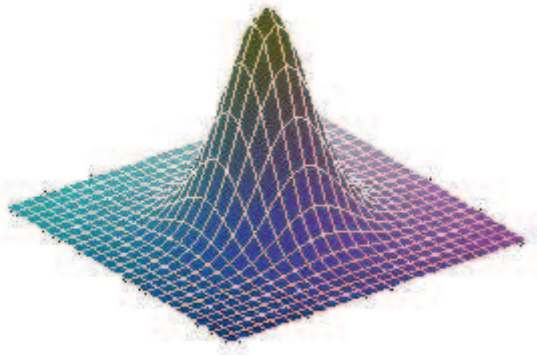

Abbildung 2.2: Mehrdimensionale Normalverteilung

Dabei wird das Originalbild mit Hilfe einer Maske gefaltet, die die Normalverteilung annähert. Der neue Grauwert eines Pixels $g_n(x,y)$ ergibt sich dabei aus den gewichteten Werten der ihn umgebenden Pixel. Ein Beispiel für eine solche Maske könnte sein:

$$\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} \quad 2.1$$

Je größer hierbei die Maske gewählt wird, desto robuster wird der Algorithmus gegenüber Rauschen. Folgen der Normalisierung kann man in Abbildung 2.3 verfolgen.

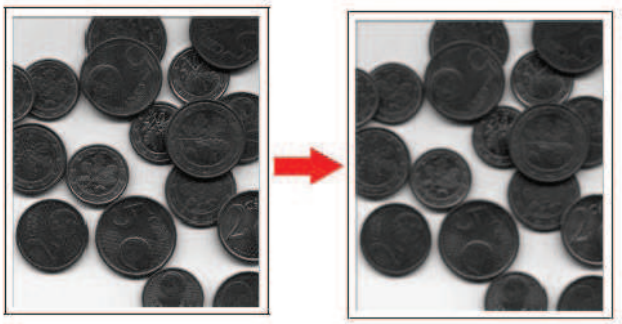

Abbildung 2.3: Normalisierung

Schritt 2.

Anschließend werden die Gradienten der einzelnen Pixel ermittelt, indem das Bild mit Hilfe des Sobeloperators gefaltet wird. Dieser arbeitet entweder in X-Richtung (2.2) oder in Y-Richtung (2.3) und betont somit entweder horizontale oder vertikale Kanten.

$$\begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} \qquad \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

2.2 H_x *2.3* H_y

Auch beim Sobeloperator ergibt sich der neue Wert eines Pixels aus den gewichteten Werten der ihn umgebenden Pixel. Da für den Canny-Algorithmus der Gradient in X-Richtung g_x (Abbildung 2.4) und der Gradient in Y-Richtung g_y (Abbildung 2.5) benötigt werden, ergeben sich also nach Anwendung des Sobeloperators 2 neue Bilder.

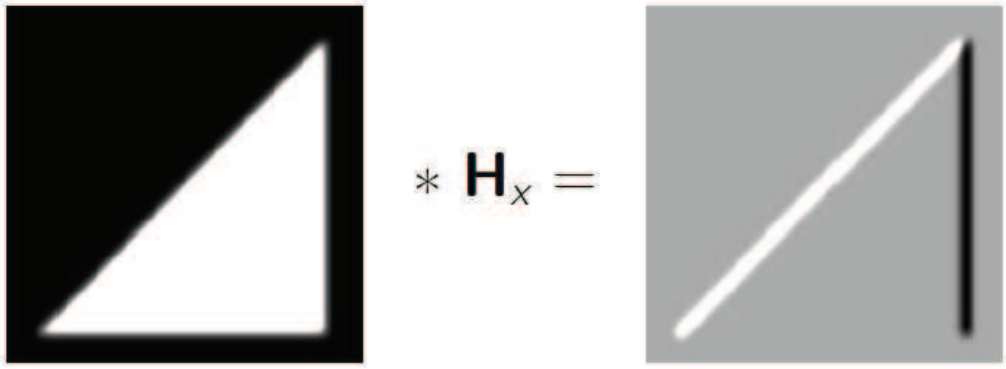

Abbildung 2.4: Ergebnis des Sobeloperators in X-Richtung

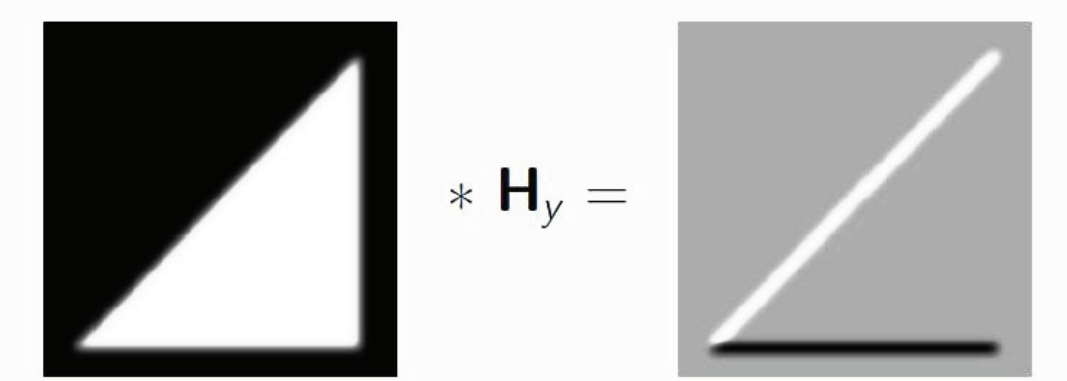

Abbildung 2.5: Ergebnis des Sobeloperators in Y-Richtung

Mithilfe der beiden so ermittelten Gradienten lässt sich der Anstieg einer potentiellen Kante durch ein Pixel leicht errechnen. Es gilt:

$$\text{Anstieg} = \begin{cases} \arctan(\frac{g_x}{g_y}), & \text{falls } g_y \neq 0 \\ 0°, & \text{falls } g_y = 0, g_x = 0 \\ 90°, & \text{falls } g_y = 0, g_x \neq 0 \end{cases}$$

2.4

Zuletzt muss noch ein Bild der absoluten Kantenstärken (Abbildung 2.6) berechnet werden.

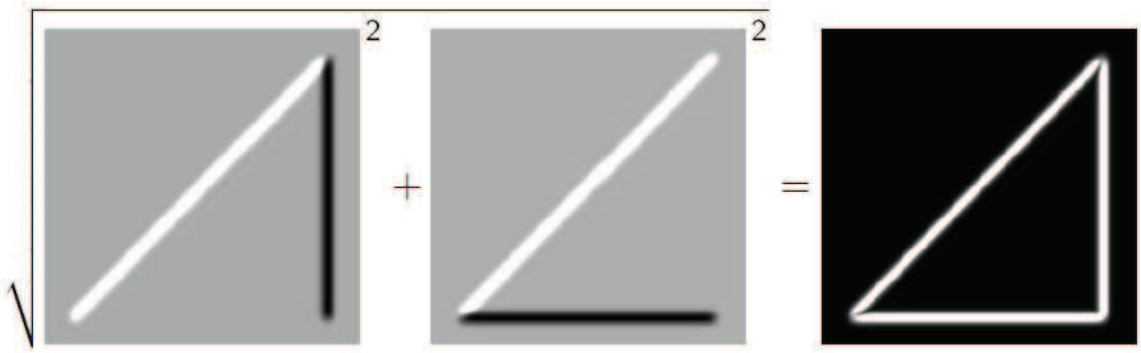

Abbildung 2.6: Berechnung der absoluten Kantenstärken

Dabei wird der Wert eines einzelnen Pixels aus dem euklidischen Betrag der beiden Gradienten gebildet (2.5).

$$G(x,y)=\sqrt{g_x(x,y)^2+g_y(x,y)^2} \qquad (2.5)$$

Schritt 3.

Um sicherzustellen, dass eine Kante nicht mehr als ein Pixel breit ist, sollen im folgenden Schritt einzig die Maxima entlang einer Kante erhalten bleiben. Dafür wird vom Bild mit den absoluten Kantenstärken ausgegangen und für jedes Pixel die Werte $G(x,y)$ mit denjenigen seiner 8 Nachbarn verglichen. Keiner der benachbarten Pixel darf einen höheren Wert aufweisen, es sei denn, der betreffende Nachbarpixel liegt entlang der berechneten Kantenrichtung. Ist dies nicht gegeben, wird der Grauwert auf Null gesetzt. Diese Technik wird "non-maximum suppression" genannt.

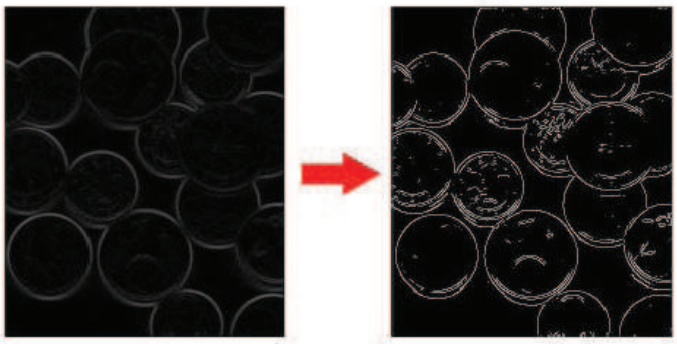

Abbildung 2.7: Non-maximum suspension

Schritt 4.

Abschließend muss natürlich noch festgestellt werden, ab welcher Kantenstärke ein Pixel zu einer Kante zu zählen ist. Um das Aufbrechen einer Kante durch Schwankungen in der errechneten Kantenstärke zu vermeiden wird ein Hysterese genanntes Verfahren angewendet. Bei diesem Verfahren verwendet man zwei Schwellwerte T1 < T2. Man scannt das Bild durch, bis ein Pixel gefunden wird, dessen Stärke größer T2 ist. Dieser Kante folgt man dann beidseitig. Alle Pixel entlang dieser Kante mit Stärke größer T1 werden als Kantenelement markiert.

Nach diesem letzten Schritt ist der Algorithmus beendet und liefert eine Menge von Punkten (Abbildung 2.8), die bei geeigneter Wahl der Schwellwerte die im Ausgangsbild vorhandenen Kanten aufzeigen.

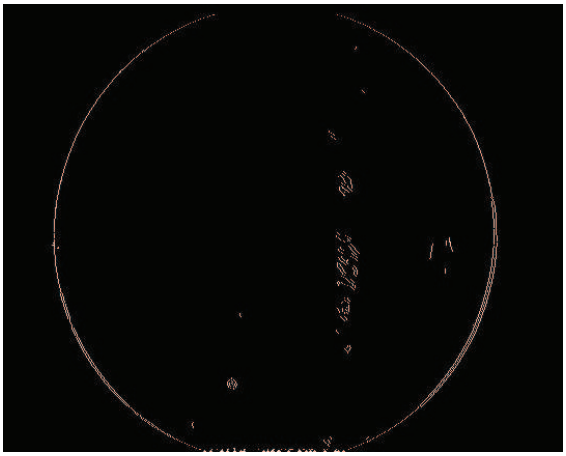

Abbildung 2.8: Kanten nach Canny Kantendetektion

2.2 Hough-Transformation

Die Hough-Transformation[9][10][11] ist ein robustes globales Verfahren zur Erkennung von Geraden, Kreisen oder beliebigen anderen parametrisierbaren geometrischen Figuren in einem binären Gradientenbild, also einem schwarz/weiß Bild, nach einer Kantenerkennung. Das Verfahren wurde 1962 von Paul V. C. Hough unter dem Namen „Method and Means for Recognizing Complex Patterns“ patentiert.

Zur Erkennung von geometrischen Objekten wird ein Dualraum erschaffen (speziell: Parameterraum, Hough-Raum), in den für jeden Punkt im Bild, der auf einer Kante liegt, alle möglichen Parameter der zu findenden Figur im Dualraum eingetragen werden. Jeder Punkt im Dualraum entspricht damit einem geometrischen Objekt im Bildraum. Bei der Geraden können das z.B. die Steigung und der y-Achsen-Abschnitt sein, beim Kreis der Mittelpunkt und Radius. Danach wertet man den Dualraum aus, indem man nach Häufungen sucht, die dann der gesuchten Figur entsprechen.

Im unseren speziellen Fall ist die Hough-Transformation für die Erkennung von Kreisen und Ellipsen interessant.

Hough-Transformation für Kreise

Ausgehend vom Kantenbild wird jedes Kantenpixel als von Kreisen eines bestimmten Radius erzeugt angesehen. Die Transformation in den Hough-Raum funktioniert so, dass man im Akkumulator alle Kreismittelpunkte einträgt, die zu diesem Kantenpixel führen könnten (jedes Akkumulatormittelpunktpixel um 1 erhöhen). Wenn nun die Punkte im Kantenbild einen Kreis repräsentieren, ist in der Akkumulatormatrix ein besonders hoher Wert an der dazugehörigen Stelle des Kreismittelpunkts eingetragen, da dort sehr viele Kantenpixel des Kreises für den Mittelpunkt abgestimmt haben. Die Maxima im Hough-Raum repräsentieren also die Kreismittelpunkte (Abbildung 2.9).

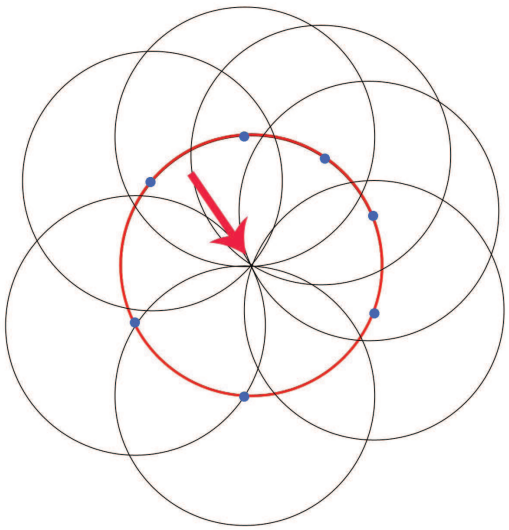

Abbildung 2.9: Häufung des Akkumulators im Mittelpunkt des Kreises

Den ersten zwei Dimensionen des Hough-Raums entsprechen in diesem Fall denen des Bildraums, da die (x,y)-Koordinaten in beiden Fällen die Lage des Kreismittelpunktes beschreiben. Zusätzlich dazu ist laut der Kreisgleichung (2.6) der Radius r der 3. Parameter, der beachtet werden muss. Man spricht bei Kreisen daher von einem 3-dimensionalen Hough-Raum (xc, yc, r). Die Wertegrenzen für den Radius müssen vorgegeben werden.

$$x^2 + x^2 = r^2$$

2.6 Kreisgleichung

Hough-Transformation für Ellipse

Bei den Ellipsen funktioniert es genauso wie bei Kreisen(Abbildung 2.11), bis auf die Tatsache, dass in der Ellipsengleichung (2.7) ein zusätzlicher Parameter auftaucht (a und b anstatt von r) (Abbildung 2.,10). Vorausgesetzt natürlich, dass die Ellipse nicht schräg liegt. (Bei unserer Aufgabe ist es nicht der Fall).

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

2.7 Ellipsengleichung

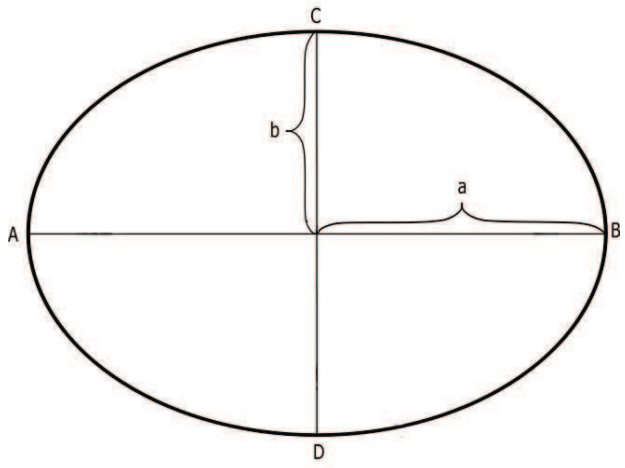

Abbildung 2.10: Ellipse

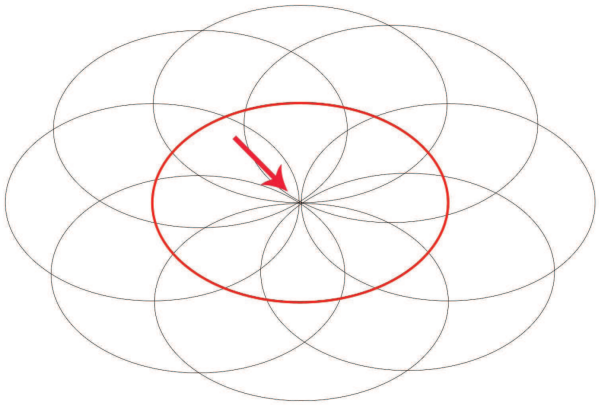

Abbildung 2.11: Häufung des Akkumulators im Mittelpunkt der Ellipse

Anschließend muss noch der Akkumulator untersucht werden. Die Zelle mit dem größten Wert repräsentiert die gesuchte Ellipse.

2.3 Verfahren von Otsu (Schwellwertverfahren)

Die Schwellwertverfahren [12] sind Algorithmen zur Segmentierung digitaler Bilder. Segmentierung allgemein kann ein wichtiger Schritt zur Bildanalyse sein, beispielsweise um Objekte im Bild zu erkennen. Mit Hilfe von Schwellwertverfahren kann man in einfachen Situationen entscheiden, welche Bildpunkte gesuchte Objekte darstellen und welche deren Umgebung angehören.

Üblicherweise binarisieren die Schwellwertverfahren ein Ausgangsbild, das heißt es werden genau zwei Segmente gebildet – im wohl häufigsten Anwendungsfall für diese Verfahren idealerweise der Hintergrund und die gesuchten Objekte. Die Zuordnung zu den beiden Segmenten (0 und 1) erfolgt aufgrund eines Vergleiches des Grauwerts g des betrachteten Pixels mit dem zuvor festgelegten Schwellwert t. Das Ergebnisbild lässt sich also mit sehr geringem Rechenaufwand berechnen, da pro Pixel nur eine einfache Vergleichsoperation durchgeführt werden muss. Die zugehörige Berechnungsvorschrift der Abbildung T_{global} lautet (2.8):

$$T_{global}(g) = \begin{cases} 0 & \text{falls } g < t \\ 1 & \text{falls } g \geq t \end{cases}$$

2.8

Unabhängig von der Wahl des Schwellwertes kann das grundlegende Prinzip der Schwellwertverfahren auf verschiedene Arten angewendet werden.

Beim **globalen Schwellwertverfahren** wird ein Schwellwert global für das gesamte Bild gewählt. Die zugehörige Berechnungsvorschrift wurde bereits oben (2.8) angegeben. Das Verfahren ist am einfachsten zu berechnen, aber auch sehr anfällig für Helligkeitsveränderungen im Bild (Abbildungen 2.12, 2.13, 2.14).

Abbildung 2.12: Das Bild mit einem Helligkeitsverlauf

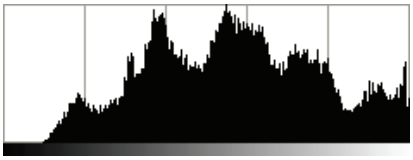

Abbildung 2.13: Das zugehörige Histogramm

Abbildung 2.14: Ergebnis der Segmentierung mit dem Schwellwert

Beim **lokalen Schwellwertverfahren** wird das Ausgangsbild in Regionen eingeteilt und der Schwellwert für jede Region getrennt festgelegt. Das bedeutet, dass in jeder Bildregion R_i ein passender Schwellwert t_i gewählt werden kann, ohne dass dies die Qualität der Segmentierung in anderen Regionen beeinflusst. Die Berechnungsvorschrift für jedes Pixel *(x,y)* lautet (2.9):

$$T_{lokal}(x,y) = \begin{cases} 0 & \text{falls } g(x,y) < t_i \\ 1 & \text{falls } g(x,y) \geq t_i \end{cases} \quad \forall (x,y) \in \text{Region } R_i$$

2.9

Gegenüber dem globalen Schwellwertverfahren steigt die Komplexität nur unwesentlich, das lokale Verfahren lässt sich daher ebenfalls mit geringem Rechenaufwand berechnen. Die Anfälligkeit gegenüber Helligkeitsveränderungen sinkt, allerdings kann es an den Grenzen der Regionen zu

Versatz kommen.

Als Weiterentwicklung des lokalen Verfahrens lässt sich das **dynamische Schwellwertverfahren** ansehen, das für jedes Pixel eine Nachbarschaft N betrachtet und auf Basis dieser Nachbarschaft einen passenden Schwellwert t(N) berechnet. Hierbei ist ein automatisches Verfahren zur Wahl des Schwellwertes zwingend notwendig. Die entsprechende Berechnungsvorschrift für jedes Pixel (x,y) lautet (2.10):

$$T_{dynamisch}(x,y) = \begin{cases} 0 & \text{falls } g(x,y) < t(N(x,y)) \\ 1 & \text{falls } g(x,y) \geq t(N(x,y)) \end{cases}$$

2.10

Der springende Punkt bei allen Schwellwertverfahren ist die Wahl eines geeigneten Schwellwertes. Dieser kann durch einen menschlichen Bearbeiter sinnvoll gewählt werden. Da beim lokalen und dynamischen Schwellwertverfahren aber eine größere Anzahl an Schwellwerten benötigt wird, bietet es sich an, ein automatisches Verfahren zur Bestimmung der Schwellwerte einzusetzen. Sowohl zur manuellen als auch zur automatischen Festlegung eines Schwellwertes ist das Histogramm das wichtigste Hilfsmittel. Lokale Maxima weisen auf die Grauwerte bzw. Grauwertbereiche hin, die im Bild vom Hintergrund oder von größeren Objekten angenommen werden.

Das **Verfahren von Otsu** [13] nach Nobuyuki Otsu aus dem Jahr 1979 verwendet statistische Hilfsmittel, um das Problem eines möglichst guten Schwellwertes zu lösen. Insbesondere wird von der Varianz Gebrauch gemacht, die ein Maß für die Streuung von Werten ist – in diesem Fall geht es um die Streuung der Grauwerte.

> Die **Varianz** ist ein Maß, das beschreibt, wie stark eine Messgröße „streut“. Sie wird berechnet, indem man die Abstände der Messwerte vom Mittelwert quadriert, addiert und durch die Anzahl der Messwerte teilt.

Das Verfahren von Otsu bestimmt einen Schwellwert, bei dem die Streuung innerhalb der dadurch bestimmten Klassen möglichst klein, zwischen den Klassen gleichzeitig aber möglichst groß ist. Otsu hat nachgewiesen, dass die Minimierung der Streuung(Varianz) innerhalb der Klassen das gleiche ist wie die Maximierung der Streuung zwischen den Klassen. Es reicht also schon, einen Schwellwert t zu finden, bei dem die Streuung(2.11) zwischen den Klassen maximal ist.

$$\sigma_b^2(t) = \omega_1(t)\omega_2(t)[\mu_1(t) - \mu_2(t)]^2$$

ω_i – *Wahrscheinlichkeit des Auftretens von Pixeln der Klassen i (Summe der Wahrscheinlichkeiten aller Gradienten)*

μ_i – *das arithmetische Mittel der Grauwerte innerhalb der Klasse i*

2.11 Varianz zwischen den Klassen

Kapitel 3

Das ImageJ-Plugin

Im folgenden Kapitel wird die Implementierung der zuvor beschriebenen Algorithmen beschrieben. Eine ausführliche Erklärung der graphischen Benutzeroberfläche schießt das Kapitel.

3.1 Die veränderten Klassen des PlugIns

Ausgehend vom ImageJ PlugIn von Martin Naderi (später hat Christian Zimmermann das PlugIn erweitert), mussten einige Klassen der Ebenen "Präsentation" und "Steuerung" leicht verändert werden.

Konkret handelt es sich um die Klassen GUI.java und Control_.java. Klasse GUI.java steht für Benutzeroberflähe. Da durch die neue Funktionalität auch neue Steuerelemente hinzugekommen sind, die an sich mit Mosaicing erst mal nichts zu tun haben, wurde ein neues ChildFrame dem DesktopFrame hinzugefügt, anstatt die neue Steuerelemente mit den alten in einem Frame zu platzieren. Der neue MaskFrame hat den Platz direkt unter dem ControlFrame gefunden (Abbildung 3.1).

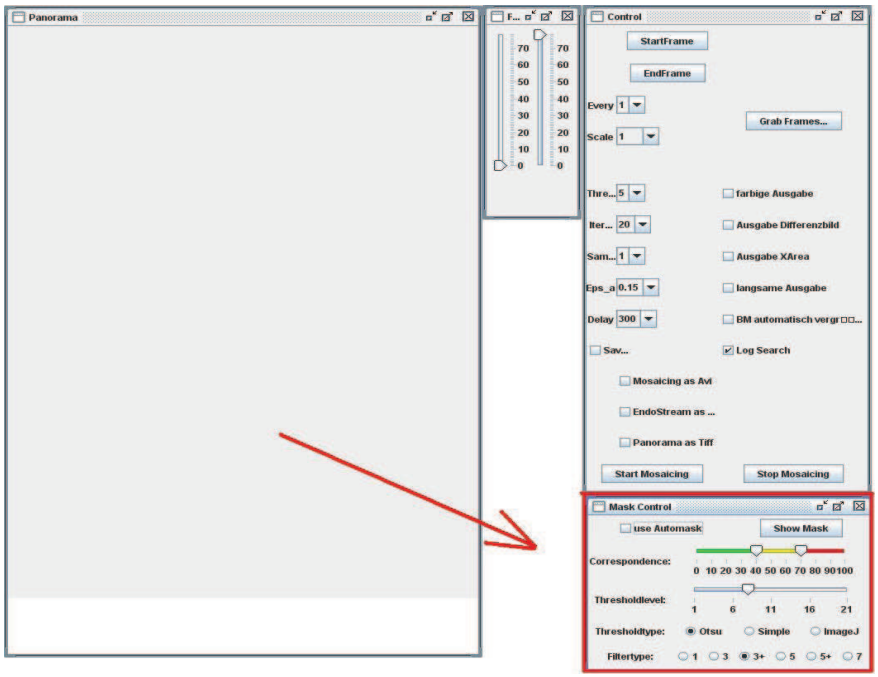

Abbildung 3.1: Die grafische Benutzeroberfläche des PlugIns

Des Weiteren wurde auch die interne Klasse GUIListener erweitert. In der Klasse ist das Verhalten der Steuerelemente programmiert. Dementsprechend wurden neue Steuerelemente mit bestimmten Reaktionen verbunden.

In der Klasse Contol_.java wurde erstens die Möglichkeit eingeräumt die automatische, - anstatt der manuellen Maske zu verwenden. Zweitens wurde eine neue Funktion angelegt, mit der man die Maskenerstellung ausgiebig testen kann.

2 neue Klassen sind angelegt worden. Klasse Filter.java implementiert die Struktur eines Filters (Matrix) und realisiert alle nötigen mathematischen Berechnungen, die durch Benutzen von Filtern verrichtet werden müssen. In der Klasse MaskCreator.java wurden alle oben beschriebenen Algorithmen, sowie auch die Steuerung über den Prozess der Maskenerzeugung implementiert. Die Methode createMask der Klasse MaskCreator ist die einzige Schnittstelle zu dem restlichen System und sie liefert die Maske (ImagePlus) als Rückgabe.

3.2 Vorgehensweise des PlugIns

Nach Untersuchung des vorhandenen Bildmaterials konnten folgende Eigenschaften festgestellt

werden:

1. Nutzbereich ist immer ellipsenförmig (fast kreisförmig)
2. Innerhalb des kreisförmigen Nutzbereichs kann ein dreieckiger Hintergrundteil direkt am Rand des Kreises vorkommen (Abbildung 3.2).
3. Dreieckiger Hintergrundteil beträgt weniger als 1% von der Gesamtfläche der Ellipse
4. Beleuchtung ist nicht gleichmäßig, so kann zum Beispiel ein Teil des Nutzbereichs genauso dunkel wie der Hintergrund sein (Abbildung 3.3).
5. Center des Kreises des Nutzbereichs liegt mehr oder weniger in der Mitte des Bildes.
6. Teil des Kreises des Nutzbereichs kann außerhalb des sichtbaren Bereichs liegen.
7. Rand des Kreises kann ein Rahmen haben. Also sich deutlich vom Hintergrund abheben, aber dennoch keine Nutzinformationen enthalten (Abbildung 3.4).
8. Hintergrund ist dunkel (Farben im Bereich 0-15), Nutzbereich ist eher hell.

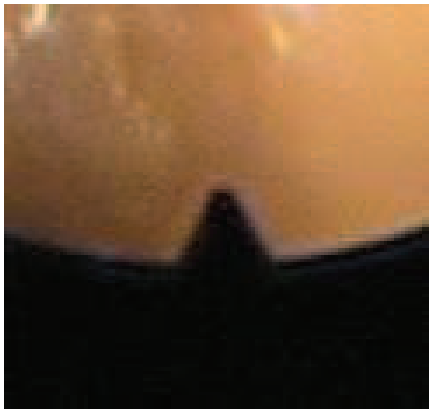

Abbildung 3.2: Dreieck im Nutzbereich

Abbildung 3.3: Verschmelzung der Bereiche

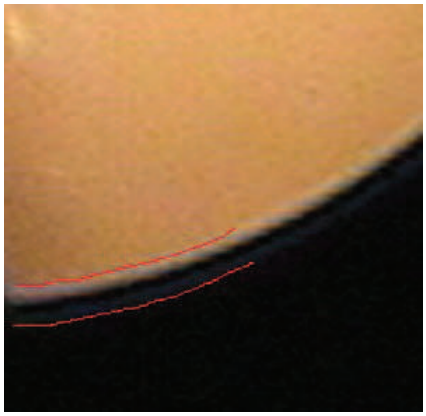

Abbildung 3.4: Rahmen

Aufgrund dieser Eigenschaften wurde folgende Vorgehensweise gewählt:

1. Im ersten Bild Kanten mit Canny-Kantendetektion detektieren
2. Anhand der Kanten einen Kreis mit Hough-Transformation finden
3. Eine Ellipse mit Hough-Transformation finden. Die Parameter des Kreises schränken die Suche nach der Ellipse ein. (Ellipsensuche ist sehr rechenaufwändig)

4. Innerhalb der Ellipse mit lokalen Schwellwertverfahren nach Dreieck suchen
5. Schritt 4 mit verschiedenen Bilder ausführen.

Die Kreissuche erfolgt relativ schnell. Dagegen kann die Ellipsensuche auch einen modernen Rechner an die Grenzen treiben. Obwohl der Kreis nicht 100% dem gesuchten Nutzbereich entspricht, man kann den gefundenen Radius für die Einschränkung von Bereich der großen und kleinen Halbachse verwenden.

Man konnte theoretisch mit Geradenerkennung mittels Hough-Transformation auch den Dreieck im Nutzbereich finden. Aber praktisch ist das nicht realisierbar, weil die Linien des Dreiecks relativ kurz sind. Tests haben gezeigt, dass durch diese Transformation etliche andere Geraden zuerst gefunden werden, selbst dann, wenn alle Kanten, die mit Sicherheit nicht zu dem Dreieck gehören vorher gelöscht werden. (Kanten der Ellipse und etliche Kanten in der Mitte des Nutzbereichs). Hinzu kommt die Tatsache, dass der Teil mit dem Dreieck auch unterbelichtet sein kann, sodass in dem Teil gar keine Geraden erkannt werden.

Aufgrund der Eigenschaft 6 muss die große und kleine Halbachse der Ellipse um 2% verringert werden, um mit Sicherheit den Rahmen auszuschließen.

In Schritt 4 wird lokales Schwellwertverfahren angewendet. Das Bild wird in Bereiche geteilt und für jeden Bereich wird ein eigener Schwellwert berechnet. Die Anzahl der Bereiche lässt sich flexibel über Benutzeroberfläche einstellen. Dabei ist es garantiert, dass es in jedem Bereich sowohl ausreichend viele Teile von Nutzbereich als auch vom Hintergrundbereich gibt. Das wird dadurch erreicht, dass es schon vorher durch die Ellipsensuche zumindest grob bekannt ist, wo der Nutzbereich ist.

Die Fläche von Dreieck beträgt maximal 1% von der Ellipsenfläche. Also wird die Suche nach dem Bereich, der dem Dreieck gehören konnte, mindestens solange fortgesetzt, bis die Fläche vom diesem Bereich mindestens 1% der Ellipsenfläche beträgt oder kein Bildmaterial mehr vorhanden ist.

Für Testzwecke ist ein Button "Show Mask" vorgesehen. Beim Anklicken des Buttons wird nur die Maske Erzeugt ohne Image Mosaicing auszuführen. Fürs ausgiebiges Testen der Maskenerzeugung ist es das wichtigste Steuerelement. Dabei kann man 3 Debugmodi unterscheiden. Die

Informationen, die je nach Modi ausgegeben werden reichen einfachen Textmitteilungen bis zu visualisierten Darstellungen des Houghraums. Textinformationen werden im Logfenster ausgegeben. Im Folgenden sind die Modi genauer beschrieben:

Debugmodus 0 - Nur die Ergebnisse der Maskenerzeugung werden bekannt gegeben:

- Bei Vorhanden sein der manuell erzeugten Maske (M) werden Statistiken ausgegeben, die aus dem Vergleich der M und automatisch erzeugten Maske (A) hervorgehen (Abbildungen 3.5, 3.6 oben links):
  - Prozentzahl der Entsprechung A und M
  - Differenz M - A (Pixel, die falsch als Hintergrund erkannt wurden)
  - Differenz A - M (Pixel, die falsch als Nutzbereich erkannt wurden) (Soll 0 sein)
- Bild mit blau markiertem Hintergrund (Abbildung 3.5)
- Bild mit blau markiertem Nutzbereich (Abbildung 3.6)
- Anzahl der eingelesenen Bilder und Anzahl der Pixel, für die keine eindeutige Entscheidung getroffen werden konnte

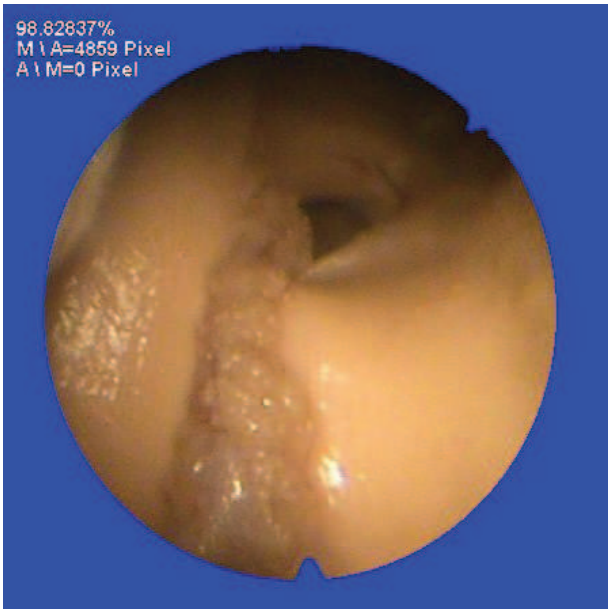

Abbildung 3.5: Bild mit blau markiertem Hintergrund

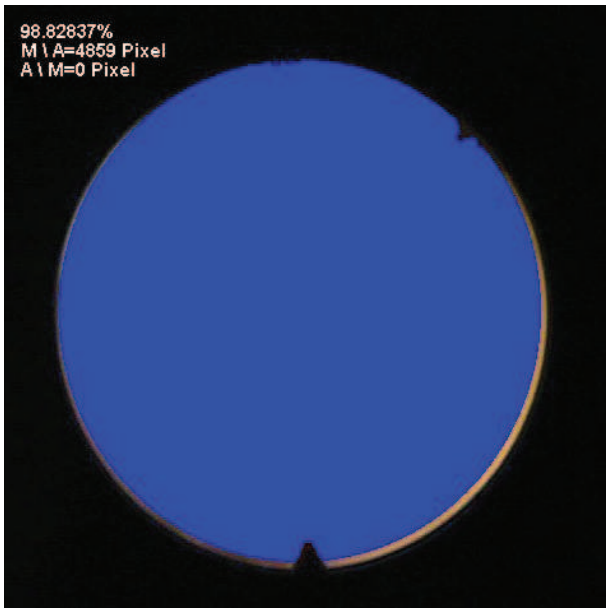

Abbildung 3.6: Bild mit blau markiertem Nutzbereich

Debugmodus 1 - Einige Zwischenschritte werden bekannt gegeben zusätzlich zu Debugmodus 0:

- Kantenbild mit eingezeichnetem Kreis(blau) und Ellipse(rot) (Abbildung 3.7)
- Originalbild mit eingezeichneter Ellipse (Abbildung 3.8)
- Länge des erkannten Radius des Kreises in Pixel
- Länge der kleinen und großen Halbachse in Pixel sowie Fläche der erkannten Ellipse
- Name von zuletzt eingelesenem Bild, Anzahl Pixel, die in dem Bild eindeutig erkannt wurde und Prozentzahl der Pixel(von Ellipsenfläche), die dem Dreieck gehören konnten.
- Anzahl der Pixel, deren Farbwert sich nicht von den Pixeln der manuellen Maske unterscheidet und Gesamtzahl der Pixel (müssen möglichst gleich sein)

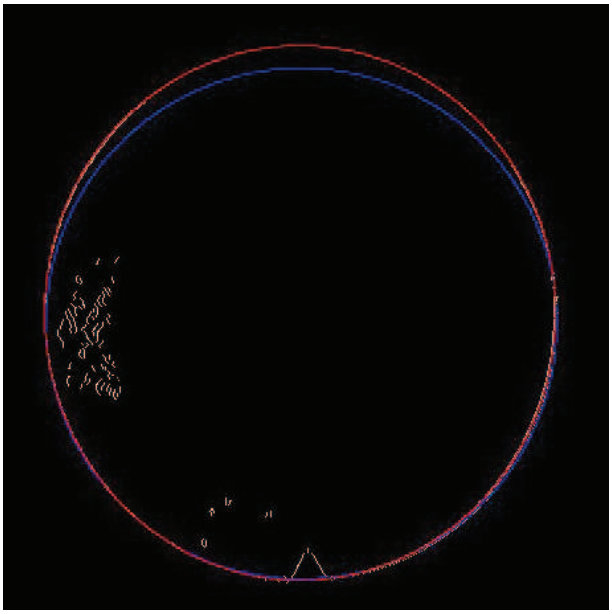

Abbildung 3.7: Kantenbild mit eingezeichneten Kreis(blau) und Ellipse(rot)

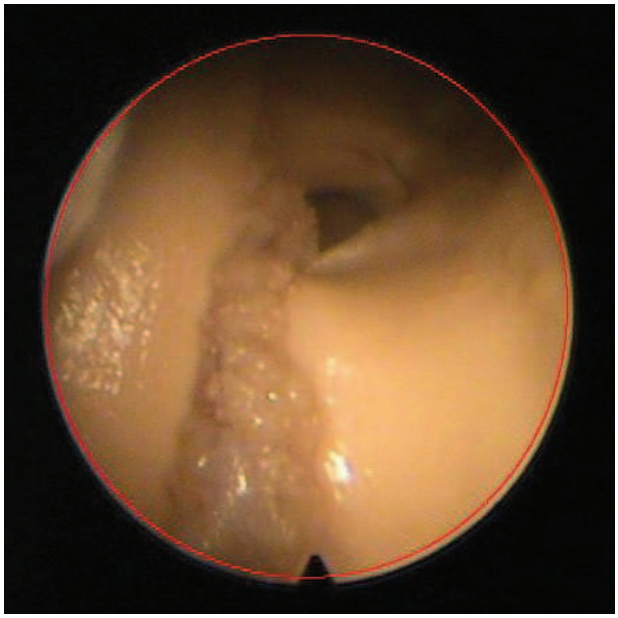

Abbildung 3.8: Originalbild mit eingezeichneter Ellipse

Debugmodus 2 - Alle Zwischenschritte werden bekannt gegeben. Zusätzlich zu Debugmodus 1:

- Houghraum von Kreissuche als Bild (Abbildung 3.9)
- Houghraum von Ellipsensuche als Bild (Abbildung 3.10)
- Die erzeugte Maske als Bild
- Koordinaten und Farbwert der Pixel von der automatisch erstellten Maske, die sich von den entsprechenden Pixel der manueller Maske unterscheiden
- Der gefundene Schwellwert

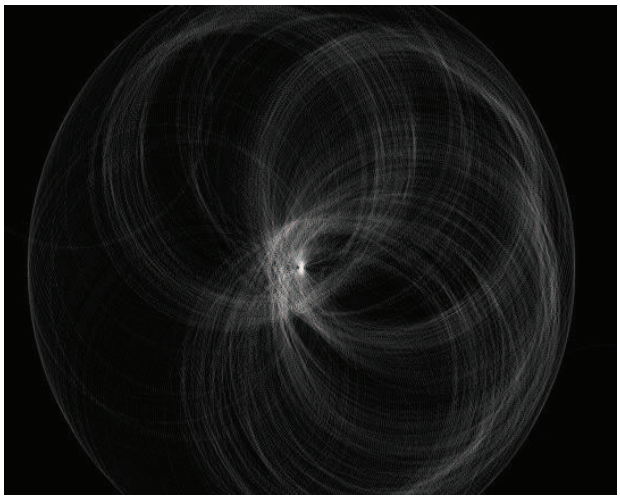

Abbildung 3.9: Houghraum von Kreissuche

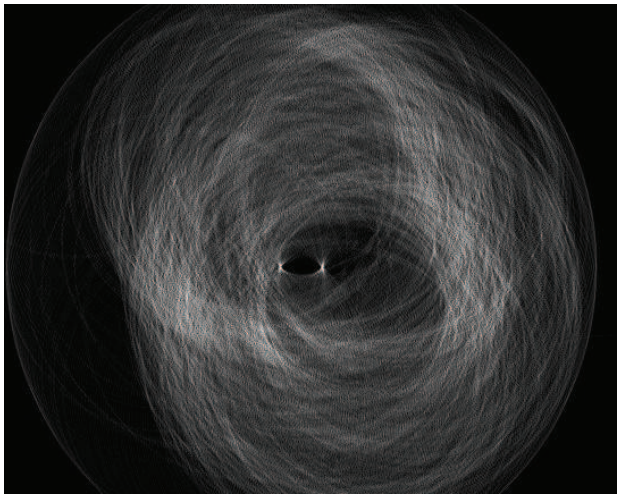

Abbildung 3.10: Houghraum von Ellipsensuche

3.3 Methodenbeschreibungen

In dem Abschnitt folgt die Beschreibung der Implementierung der oben beschriebenen Algorithmen sowie der Steuerung über den Ablauf. Eine detaillierte Beschreibung hilft dabei, nachfolgende Arbeiten am PlugIn zu erleichtern.

Hinweis: In den folgenden Listings sind bewusst einige Abweichungen vom Originalquellcode enthalten. Das ist extra so gemacht, damit man sich nicht durch die Variablendeklarationen oder das Sammeln von statistischen Daten ablenken lässt.

3.3.1 MaskCreator.findThresholdNachOtsu

Die Methode wird zur Berechnung von Schwellwert aufgerufen. Als Parameter wird Histogramm von Typ Integer Array übergeben. Otsu schlägt vor, solche zwei Klassen von Punkten zu finden, dass die Varianz zwischen den Klassen am größten und Varianz in den Klassen am kleinsten ist. Dabei reduziert sich die Aufgabe auf das Finden der größten Varianz zwischen den Klassen, weil Otsu bewiesen hat, dass die größte Varianz zwischen den Klassen gleichzeitig kleinste Varianz in den Klassen bedeutet. Noch einmal zur Erinnerung die Formel der Varianz zwischen den Klassen:

$$\sigma_b^2(t) = \omega_1(t)\,\omega_2(t)\,[\mu_1(t) - \mu_2(t)]^2$$

ω_i – *Wahrscheinlichkeit des Auftretens von Pixeln der Klassen i*
(Summe der Wahrscheinlichkeiten aller Gradienten)
μ_i – *das arithmetische Mittel der Grauwerte innerhalb der Klasse i*
3.1 Varianz zwischen den Klassen

Dabei sind ω_i und μ_i unbekannt und müssen vorher berechnet werden. Wahrscheinlichkeit des Auftretens von Pixel der Klasse ist die Summe der Wahrscheinlichkeiten des Auftretens der Grauwerte aus der Klasse. In Listing 3.1 werden zuerst die Wahrscheinlichkeiten der Grauwerte berechnet (Zeilen 13-14). Und anschließend werden die Wahrscheinlichkeiten der Klassen zusammen mit dem arithmetischen Mittel der Klasse in der Schleife in Zeilen 15-30 berechnet. Die Summe der Wahrscheinlichkeiten, Summe der Grauwerte und Anzahl der Grauwerte für Klasse 0 werden in interner Schleife in Zeilen 17-21 und für Klasse 1 in der Schleife in Zeilen 24-28 berechnet. In Zeile 22 für Klasse 0 und in Zeile 29 für Klasse 1 wird das jeweilige arithmetische Mittel berechnet. In der Schleife in Zeilen 31- 37 wird unmittelbar die Varianz zwischen den Klassen berechnet und Gleichzeitig das Maximum gesucht.

```
01    int numberOfPixels = 0;
02    float temp=0, temp2=0;
03    int result=0;
04    float q=0;
05    float [] wahrscheinlichkeit = new float [histo.length];
06    float [][] w = new float [histo.length][2];
07    float [][] m = new float [histo.length][2];
08    for (int i=0; i<histo.length; i++){
09          numberOfPixels+=histo[i];
10          w[i][0]=0; w[i][1]=0;
11          m[i][0]=0; m[i][1]=0;
12    }
13    for (int i=0; i<histo.length; i++)
14          wahrscheinlichkeit[i]=histo[i]/numberOfPixels;
15    for (int t=0; t<histo.length; t++){
16          temp=0;temp2=0;
17          for (int i=0; i<=t; i++){
18                w[t][0]+=wahrscheinlichkeit[i];
19                temp+=histo[i]*i;
20                temp2+=histo[i];
21          }
22          m[t][0]=temp/temp2;
23          temp=0;temp2=0;
24          for (int i=t+1; i<wahrscheinlichkeit.length-1; i++){
25                w[t][1]+=wahrscheinlichkeit[i];
26                temp+=histo[i]*i;
27                temp2+=histo[i];
28          }
29          m[t][1]=temp/temp2;
30    }
31    for (int t=1; t<histo.length; t++){
32          temp=w[t][0]*w[t][1]*(m[t][0]-m[t][1])^2);
33          if (q<temp) {
34                q=temp;
35                result=t;
36          }
37    }
```

Listing 3.1: Schwellwertverfahren nach Otsu

3.3.2 MaskCreator.findThreshold

Das ist der von mir selbst entworfenes Algorithmus, der in manchen Fällen besser als die anderen funktioniert. Er basiert auf der Annahme [7], dass das kleinste Element zwischen 2 Maxima der gesuchte Schwellwert ist. Der Quellcode vom Algorithmus ist im Listing 3.2 zu finden. Aufgrund der Eigenschaft 7 (Hintergrund ist dunkel (Farben im Bereich 0-15), Nutzbereich ist eher hell) kann man die Annahme treffen, dass das lokale Maximum für Hintergrund in dem ersten Fünftel des Histogramms zu finden ist (Zeile 06). Nach dem Finden des größten Wertes (Zeile 07)

und dessen Index (Zeilen 08-09) in dem ersten Fünftel, wird das lokale Minimum nach rechts vom ersten Maximum gesucht , sodass es mindestens 4 Elemente rechts von dem größer als es selbst sein müssen (Zeilen 11-16). Ab dem lokalen Minimum wird dann wieder nach rechts und bis zum Ende nach dem zweiten Maximum gesucht (Zeilen 17-19). Anschließend wird noch einmal nach Minimum zwischen zwei Maxima gesucht (Zeilen 20-22).

```
01    int max1; //erstes lokales Maximum
02    int maxIndex1=0, maxIndex2; //Index von Maxima
03    int minIndex; //Index von Minimum
04    int lookUp=4;
05
06    int[] temp = Arrays.copy(histogramm, 0, histogramm.length/5-1);
07    max1 = Arrays.max(temp);
08    for (int i=0; i<temp.length;i++)
09          if (histogramm[i]==max1) {maxIndex1=i; break;}
10
11    minIndex=maxIndex1;
12    for (int i=1; i<=lookUp; i++)
13          if (histogramm[minIndex]>histogramm[minIndex+i]) {
14                minIndex+=i;
15                i=0;
16          }
17    maxIndex2=minIndex;
18    for (int i=minIndex; i<histogramm.length;i++)
19          if (histogramm[i]>histogramm[maxIndex2]) maxIndex2=i;
20    minIndex = maxIndex1;
21    for (int i=maxIndex1+1; i<maxIndex2; i++ )
22          if (histogramm[i]<histogramm[minIndex]) minIndex=i;
```

Listing 3.2: Schwellwertverfahren

3.3.3 MaskCreator.cannyA

Diese Methode realisiert die Canny Kantendetektion. Als Parameter wird nur ImageProcessor übergeben. Es verläuft in 4 Schritten bis daraus ein sauberes Kantenbild entsteht:

Schritt 1 (Listing 3.3)**:** Glättung mit dem Gauss-Filter. In Zeile 1 wird der Gauss-Filter erzeugt und in der 2-ten wird er auf das ganze Bild angewendet

```
01    Filter gauss = new Filter({{1,2,1},{2,4,2},{1,2,1}});
02    gauss.applyForImage(img);
```

Listing 3.3: Glättung mit Gauss-Filter

Schritt 2 (Listing 3.4): Betonung der Kanten und Berechnen des Anstiegs der Pixel der Kanten. In den Zeilen 02 und 03 werden die Sobeloperatoren in X und Y-Richtung erzeugt. In den Zeilen 05

und 06 werden sie auf das ganze Bild separat angewendet und die Ergebnisse in Integerarrays gespeichert. In der Schleife in Zeile 11 werden die beiden berechneten Gradienten in ein Bild der absoluten Kantenstärke zusammengeführt. Dabei wird der Wert eines einzelnen Pixels aus dem euklidischen Betrag der beiden Gradienten gebildet. In Zeile 14 wird mit Hilfe des Arkustangens der Anstieg der einzelnen Pixel berechnet. Da es mit der 8-ter Nachbarschaft gearbeitet wird, werden die gefundenen Anstiege auf 0, 45, 90 oder 135 Grad in Zeilen 12 bis 20 abgerundet.

```
01    byte [][] anstieg = new char [img.getWidth()][img.getHeight()];
02    Filter fx = new Filter({1,0,-1},{2,0,-2},{1,0,-1});
03    Filter fy = new Filter({1,2,1},{0,0,0},{-1,-2,-1});
04
05    int[][] gx = fx.applyForImage(img.getIntArray());
06    int[][] gy = fy.applyForImage(img.getIntArray());
07
08    double alfa;
09    for (int x=0; x<img.getWidth();x++)
10          for(int y=0; y<img.getHeight(); y++){
11                img.set(x,y,Math.sqrt(gx[x][y]^2 + gy[x][y]^2));
12                if (gy[x][y]!=0){
13                      alfa = Math.atan(gx[x][y]/gy[x][y])*180/Math.PI;
14                      if      (alfa<-67.5) anstieg[x][y] = 90;
15                      else if (alfa<-22.5) anstieg[x][y] = 135;
16                      else if (alfa< 22.5) anstieg[x][y] = 0;
17                      else if (alfa< 67.5) anstieg[x][y] = 45;
18                      else                 anstieg[x][y] = 90;
19                }else if (gx[x][y]!=0)     anstieg[x][y] = 90;
20                 else                      anstieg[x][y] = 0;
21          }
```

Listing 3.4: Kantenbetonung und Anstiegsberechnung

Schritt 3 (Listing 3.5): Verdünnung der Kanten. Um sicher zu stellen, dass die Kante maximal 1 Pixel breit ist, wird für jedes Pixel mit denjenigen seiner 8 Nachbarn verglichen. In Zeilen 05-07 werden alle Nachbarwerte abgefragt und in den Variablen G1-G8 gespeichert. Keiner der benachbarten Pixel darf einen höheren Wert aufweisen, es sei denn, der betreffende Nachbarpixel liegt entlang der berechneten Kantenrichtung. Ist dies nicht gegeben, wird der Grauwert auf Null gesetzt. (Zeilen 08-17).

```
01    int[][] temp=img.getIntArray();
02    int G,G1,G2,G3,G4,G5,G6,G7,G8;
03    for (int x=1; x<img.getWidth()-1;x++)
04          for(int y=1; y<img.getHeight()-1; y++){
05                G =temp[x  ][y  ]; G1=temp[x-1][y-1]; G2=temp[x  ][y-1];
06                G3=temp[x+1][y-1]; G4=temp[x-1][y  ]; G5=temp[x+1][y  ];
07                G6=temp[x-1][y+1]; G7=temp[x  ][y+1]; G8=temp[x+1][y+1];
08                switch (anstieg[x][y]){
09                      case 0:   if (G1>G || G2>G || G3>G || G6>G || G7>G || G8>G)
10                                      img.set(x,y,0); break;
11                      case 45:  if (G1>G || G2>G || G4>G || G5>G || G7>G || G8>G)
12                                      img.set(x,y,0); break;
13                      case 90:  if (G1>G || G3>G || G4>G || G5>G || G6>G || G8>G)
14                                      img.set(x,y,0); break;
15                      case 135: if (G2>G || G3>G || G4>G || G5>G || G6>G || G7>G)
16                                      img.set(x,y,0); break;
17                }
18          }
```

Listing 3.5: Verdünnung der Kanten

Schritt 4 (Listing 3.6): Binarisieren mit Schwellwert. Es wird ein Schwellwert T2 gefunden (Zeile 02). Aus vielen Tests ist ersichtlich geworden, dass sich für die Aufgabe am besten Schwellwertverfahren von Otsu eignet. Und zwar, wenn man von dem nach Otsu berechneten Wert noch 10 abzieht (Zeile 02). Ausgehend von T2 wird ein kleinerer Schwellwert T1 gewählt (Zeile 03). Für jedes Pixel wird geprüft, ob es größer T2 ist (Zeile 06). Ist das der Fall, so wird der Wert auf 255 (weiß) gesetzt und auch geprüft, ob die Nachbarpixel in Kantenrichtung größer T1 sind (Zeilen 11-20). In dem Fall werden auch die auf weiß gesetzt. Anschließend werden alle Pixel, die nicht weiß sind auf 0 (schwarz) gesetzt. Und so bekommen wir ein sauberes Kantenbild. Das Bild wird am Ende zurückgegeben.

```
01    int[][] temp=img.getIntArray();
02    int T2 = findThresholdNachOtsu(img.getHistogram())-10;
03    int T1=T2-20;
04    for (int x=1; x<img.getWidth()-1;x++)
05          for(int y=1; y<img.getHeight()-1; y++)
06                if (img.get(x,y)>T2){
07                      img.set(x,y,255);
08                      G1=temp[x-1][y-1];G2=temp[x  ][y-1];G3=temp[x+1][y-1];
09                      G4=temp[x-1][y  ];G5=temp[x+1][y  ];G6=temp[x-1][y+1];
10                      G7=temp[x  ][y+1];G8=temp[x+1][y+1];
11                      switch (anstieg[x][y]){
12                            case 0:   if (G4>T1) img.set(x-1,y  ,255);
13                                      if (G5>T1) img.set(x+1,y  ,255); break;
14                            case 45:  if (G3>T1) img.set(x+1,y-1,255);
15                                      if (G6>T1) img.set(x-1,y+1,255); break;
16                            case 90:  if (G2>T1) img.set(x  ,y-1,255);
17                                      if (G7>T1) img.set(x  ,y+1,255); break;
18                            case 135: if (G1>T1) img.set(x-1,y-1,255);
19                                      if (G8>T1) img.set(x+1,y+1,255); break;
20                      }
21                }
22    for (int x=0; x<img.getWidth();x++)
23          for(int y=0; y<img.getHeight(); y++)
24                if (img.get(x,y)!=255) img.set(x,y,0);
```

Listing 3.6: Binarisieren mit Schwellwert

3.3.4 MaskCreator.calculateHoughSpaceForEllipse

Diese Methode berechnet den Hough Akkumulator für Ellipsen mit vorgegebenen kleinen und großen Halbachsen. Es lässt sich auch für Kreise anwenden. Allerdings sind beim Kreis große und kleine Halbachse gleich. Die Werte für Halbachsen und Image selbst werden als Parameter übergeben. Methode ist in Listing 3.7 angeführt. Zuerst wird ein Akkumulator angelegt, der dieselben Abmessungen wie das Bild hat (Zeile 02) und er wird mit 0 initialisiert (Zeilen 03-05). Anschließend wird für jedes Pixel auf der Kante (=255) eine Ellipse mit vorgegebenen Werten für große(b) und kleine Halbachse(a) um das Pixel herum "gezeichnet". Da die Ellipse in X und Y-Richtung symmetrisch ist, reicht es die Koordinaten für die Punkte der Ellipse aus einem Viertel zu berechnen (Zeile 10) und zwar von 0 bis 90 Grad. Alle anderen Koordinaten lassen sich, durch die Symmetrie bedingt, leicht berechnen (Zeile 14-16). Dabei wird noch geprüft, ob Koordinaten außerhalb von Akkumulator fahlen. In dem Fall werden diese Pixel einfach ignoriert. Anschließend wird der Akkumulator zurückgegeben.

```
01    int dX,dY;
02    int[][] akku= new int[width][height];
03    for (int x=0; x<width;x++)
04          for(int y=0; y<height; y++)
05                akku[x][y] = 0;
06
07    for (int x=0; x<width;x++)
08          for(int y=0; y<height; y++)
09                if (img.get(x,y)==255)
10                      for (int α=0; α<=90; α++){
11                            dX=(int)(Math.cos(α*Math.PI/180)*a);
12                            dY=(int)(Math.sin(α*Math.PI/180)*b);
13                            if (x+dX<width && y+dY<height)akku[x+dX][y+dY]++;
14                            if (x-dX>=0    && y+dY<height)akku[x-dX][y+dY]++;
15                            if (x-dX>=0    && y-dY>=0)    akku[x-dX][y-dY]++;
16                            if (x+dX<width && y-dY>=0)    akku[x+dX][y-dY]++;
17                      }
18    return akku;
```

Listing 3.7: Berechnung von Hough Akkumulator für Ellipse und Kreise

3.3.5 MaskCreator.findEllipse(ImagePlus img, int houchModus)

Diese Methode versucht in dem als Parameter übergebenen Bild eine Ellipse zu finden und ihre Parameter (Mittelpunktkoordinaten sowie Länge der großen und kleinen Halbachse) zu speichern. Da die Ellipsensuche sehr rechenaufwändig ist, wird versucht, die Suche so weit wie möglich einzuschränken. Zuerst wird nach einem Kreis gesucht. Zwar kann man mit bloßem Auge sehen, das der Kreis der Kontur nur wenig entspricht (Abbildung 3.11, 3.12), aber den gefundenen Radius kann man in der Ellipsensuche weiterverwenden.

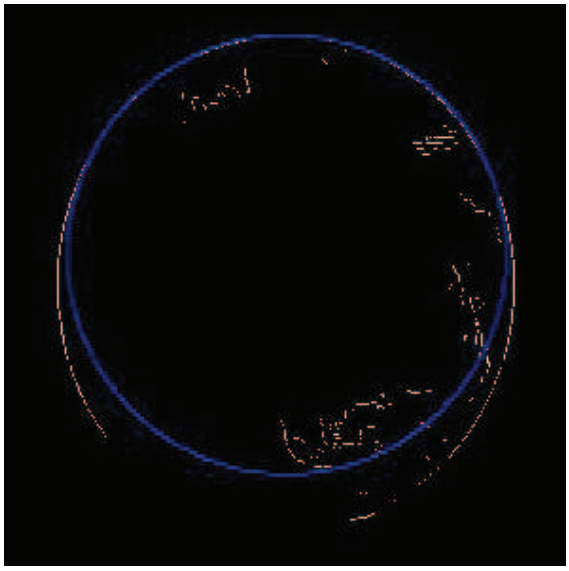

Abbildung 3.11: Kreiserkennung I

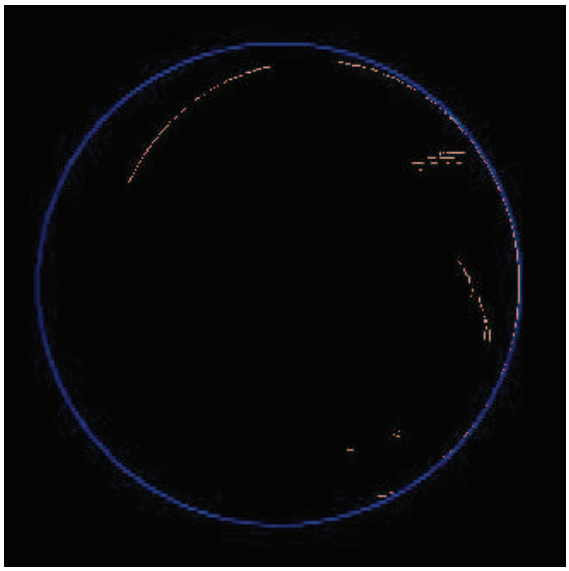

Abbildung 3.12: Kreiserkennung II

Die Kreissuche ist in Listing 3.8 aufgeführt. Zuerst wird ein Kantenbild mit Canny-Kantendetektion erzeugt (Zeile 1). Dann wird für jeden Radiuswert ein Houghraum erzeugt(Zeile 06) und untersucht (Zeilen 07-13). Wird im Akkumulator ein größerer Wert gefunden als bisher bekannt, so wird er, der Radius und die Koordinaten vom diesem Mittelpunkt gespeichert (Zeilen 09-13). Der Houghraum mit dem größten Wert wird ebenfalls gespeichert (Zeile 12) und im Debugmodus 2 visualisiert dargestellt.

```
01    cannyA(img);
02    int[][] houghraumKreis, houghraumKreisBest=null;
03    int maxCircle=0,xBestC=0,yBestC=0,rBest=0;
04
05    for (int r=height/4; r<height*4/7; r++){
06          houghraumKreis = calculateHoughSpaceForEllipse(temp,r,r);
07          for (int x=0; x<width;x++)
08                for(int y=0; y<height; y++)
09                      if(houghraumKreis[x][y]>maxCircle){
10                            maxCircle=houghraumKreis[x][y];
11                            xBestC=x; yBestC=y; rBest=r;
12                            houghraumKreisBest=houghraumKreis;
13                      }
14    }
```

Listing 3.8: Kreissuche

Wie schon im Kapitel 3.2 erklärt, ist die Ellipse nur ein wenig abgeplattet im Vergleich zu dem Kreis. So kann man davon ausgehen, dass sich die große und die kleine Halbachse maximal um 20% vom Radius unterscheiden. Auf diese Weise kann man die Wartezeiten erheblich reduzieren.

Darüber hinaus werden weitere Tricks verwendet, um die Wartezeit zu reduzieren. So kann man in der Benutzeroberfläche zwischen 3 Houghmodi wählen. Die Suche nach der Ellipse verläuft genau so wie die Suche nach dem Kreis bis auf die Tatsache, dass bei der Ellipse 2 Parameter (sprich eine zusätzliche interne Schleifen) untersucht werden (Listing 3.9). Das ist übrigens auch die langsamste Variante und der Fall für Houghmodus "*slow*". Diese Variante liefert die besten Ergebnisse dafür muss man aber lange Wartezeiten in Kauf nehmen.

```
01    for (int b=(int)(rBest*0.8); b<rBest*1.2; b++)
02          for (int a=(int)(rBest*0.8); a<rBest*1.2; a++){
03                houghraumEllipse = calculateHoughSpaceForEllipse(img,a,b);
```

Listing 3.9 Houghmodus "slow"

Houghmodus "*medium*" versucht zuerst grob die Ellipse zu finden (Listing 3.10). Dafür werden die Werte für die große und kleine Halbachse in großen Schritten durchgegangen a=a+5, b=b+5 (Zeilen 01-02). Anschließend werden die gefundenen Werte für große und kleine Halbachse für die Einschränkung der Suche in der zweiten Schleife verwendet (Zeilen 11-12). Diese Variante liefert ebenfalls gute Ergebnisse, kann aber die Wartezeit erheblich reduzieren.

```
01    for (int b=(int)(rBest*0.8); b<rBest*1.2; b=b+5)
02          for (int a=(int)(rBest*0.8); a<rBest*1.2; a=a+5){
...
...
11    for (int b=(int)(bBest*0.95); b<bBest*1.05; b++)
12          for (int a=(int)(aBest*0.95); a<aBest*1.05; a++){
```

Listing 3.10: Houghmodus "medium"

Houghmodus "*fast*" *(Listing 3.11)* versucht zuerst nur den Wert für die kleine Halbachse zu finden (Zeilen 01-02) und geht dabei davon aus, dass die große Halbachse dem Radius gleicht. Anschließend wird nach dem Wert für die große Halbachse gesucht. Diese Variante ist fehleranfällig, kann aber die Wartezeiten viel mehr reduzieren.

```
01    for (int b=(int)(rBest*0.8); b<rBest*1.2; b++){
02          houghraumEllipse = calculateHoughSpaceForEllipse(temp,rBest,b);
...
...
11    for (int a=(int)(rBest*0.8); a<rBest*1.2; a=a+1){
12          houghraumEllipse = calculateHoughSpaceForEllipse(temp,a,bBest);
```

Listing 3.11: Houghmodus "fast"

3.3.6 Einige Hilfsmethoden

Um das nachfolgende Listing 3.13 im Kapitel 3.3.7 verstehen zu können, sind einige Hilfsfunktionen notwendig.

Die Methode **fillThreaschold** liefert eine Tabelle mit Schwellwerten in Form von Array zurück. Das hat damit zu tun, dass nach dem Dreieck innerhalb der Ellipse mit lokalen Schwellwertverfahren gesucht wird. Das Bild wird vertikal immer in 2 Bereiche geteilt. Die horizontale Anzahl der Bereiche lässt sich über Benutzeroberfläche einstellen. Es wird allerdings nicht das ganze Bild in Bereiche geteilt, sondern nur der Teil des Bildes, wo Ellipse erkannt wurde (Abbildung 3.13). Im Listing 3.12 werden zuerst die Koordinaten des höchsten und der niedrigsten Pixel von Ellipse ermittelt (Zeilen 01-02), danach wird ein Array `int[][] thresholds` in Zeile 03 angelegt, wo später die Schwellwerte reingeschrieben werden. In Zeile 04 wird die Breite von Bereich ermittelt und in der darauffolgenden Schleife werden die Schwellwerte ermittelt und in die Tabelle geschrieben. Dabei sorgt der Befehl `img.setRoi(x,y,dx,dy )` dafür, dass der darauffolgende Befehl `img.getHistogram()` Histogramm nur von dem Bereich liefert, der im Punkt (x,y) anfängt und Breite dx und Höhe dy hat.

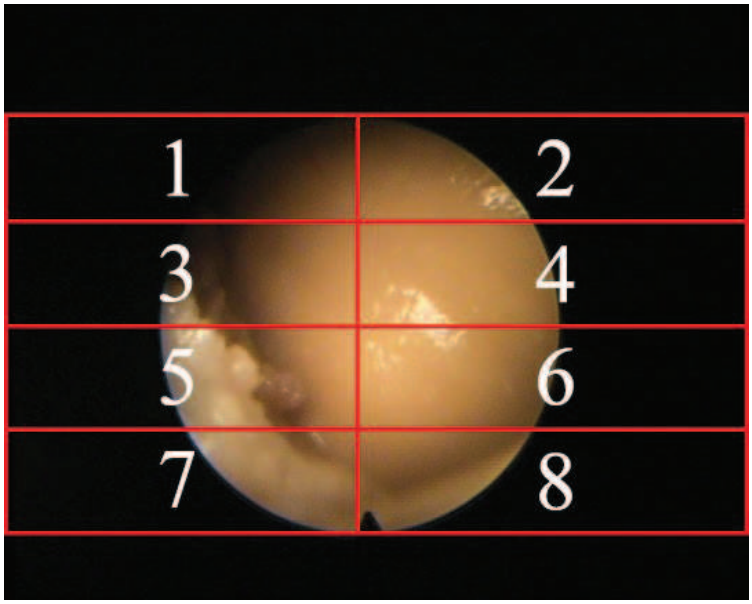

Abbildung 3.13: lokales Schwellwertverfahren

```
01    int up    = (Y-B)<0?0:Y-B;
02    int down  = (Y+B)>(img.getHeight()-1)?(img.getHeight()-1):(Y+B);
03    int[][] thresholds = new int [2][thresholdLevel];
04    int step = (down-up)/thresholdLevel;
05    for (int y=0; y<thresholdLevel; y++){
06          img.setRoi(0, y*step+up, X, step);
07          thresholds[0][y] = findThreshold(img.getHistogram());
08
09          img.setRoi(X, y*step+up, X, step);
10          thresholds[1][y] = findThreshold(img.getHistogram());
11    }
12    return thresholds;
```

Listing 3.12: Tabelle mit Schwellwerte berechnen

Die Methode ***getThreshold*** `(int[][] thresholds, int x, int y)` sucht in der Schwellwerttabelle `thresholds` nach passendem Schwellwert für Pixel mit Koordinatenn *x,y*.

Klasse ***Filter.java*** realisiert den Filter. Die Basis der Klasse bildet eine Matrix. Die Werte der Zellen kann man entweder dem Konstruktor übergeben oder auch später setzen. Außerdem gibt es in der Klasse 2 Zeiger, die auf die Mitte der Matrix zeigen (x und y Index). Wenn der Filter auf eine Stelle im Bild angelegt wird, so kommt die Mitte der Matrix (worauf die beiden Zeiger zeigen) genau auf die Stelle.

Funktion ***calculatePercentage(ImageProcessor img, int bg, int x, int y)*** berechnet wie viel Prozent der Pixel den Wert größer als `int` *bg* haben. Dabei wirken sich die Werte der Zellen der Matrix so aus, dass die größeren Werte größere Gewichtung haben.

Funktion ***apply (int[][] img, int x, int y)*** - berechnet die Summe(S1) der Werte von dem Bild multipliziert mit den Werten der Matrix. Anschließend wird die Summe geteilt durch die Summe(S2) der Werte der Zellen der Matrix. Wenn die Summe der Werte der Zellen der Matrix gleich 0 ist, dann wird die berechnete Summe(S1) direkt zurückgegeben.

Funktion ***applyForImage(ImageProcessor img)*** legt den Filter auf alle Pixel des Bildes auf. Diese Funktion kann auch als Parameter ein zweidimensionales Integer-array bekommen. Das ist dann nützlich, wenn durch die Berechnungen einige Werte größer 255 sein können.

3.3.7 MaskCreator.createMask

Das ist die wichtigste Methode, die auch Schnittstelle für die ganze Maskenerzeugung bildet. Im

Listing 3.13 ist die Quellcode aufgeführt. In Zeile 01 wird ImageProcessor für die Maske erzeugt und gleich mit 0 initialisiert in Zeilen 02-03. In Zeile 04 wird Befehl aufgerufen, der die Ellipsensuche veranlasst. Die Fläche der gefundenen Ellipse wird in dem ImageProcessor der Maske auf 255 (weiß) gesetzt (Zeilen 05-06). Dies wird später ein Signal dafür sein, dass an dieser Stelle nach Dreieck gesucht werden soll. Die Variablen in Zeilen 07-10 werden die Suche insoweit einschränken, dass wirklich nur in dem Bereich der Ellipse gesucht wird und nicht an den Rändern. In der Schleife in Zeilen 11-40 werden nach jedem Durchlauf neue Bilder geladen und konsequent untersucht. Ob ein Pixel dem Nutzbereich gehört wird in Zeile 20 entschieden. Die Variable *prozent* enthält die Prozentzahl, die mit `calculatePercentage` berechnet wurde (sehe Kapitel 3.3.6). Ist Prozentzahl aus Variable *prozent* größer als die von Benutzeroberfläche eingestellte maximale Prozentzahl (Zeile 20), so wird das Pixel als zu Nutzbereich angehörig erkannt und auf 1 in der Maske gesetzt. Ist sie kleiner als die von Benutzeroberfläche eingestellte minimale Prozentzahl (Zeile 23), so wird das Pixel als dem Hintergrund angehörig erkannt und auf 10 in der Maske gesetzt (damit man diese Pixel von den Pixel außerhalb der Ellipse unterscheiden kann). Ist sie dazwischen, so wird das Pixel als nicht eindeutig identifizierbar angesehen und zur weitere Untersuchung mit dem nächsten Bild freigelassen. Die auf 10 gesetzte Pixel (zu Hintergrund angehörig) werden in dem nächsten Schleifendurchgang auch nochmal untersucht, damit sie bei Falscherkennung (z. B. durch schlechte Beleuchtung) korrigiert werden können. Dabei werden die Pixel gezählt, die entweder dem Hintergrund angehören oder nicht eindeutig erkannt worden sind (Zeile 22). Diese Information wird gebraucht, um zu berechnen, wie viel Pixel konnte den Dreieck gehören. Ist diese Zahl größer als 1% der ganzen Ellipsenfläche, so wird weiter gesucht (Zeile 31). Außerdem ist Vorhandensein der nicht eindeutig erkannten Pixel auch ein Indiz dafür, weiter zu suchen (Zeilen 28-30). Wenn es bis Zeile 32 entschieden wurde weiter zu suchen, so wird in Zeilen 32-39 versucht das nächste Bild zu laden. Dabei gelten folgende Bedienungen:

- Nach dem zehnten Bild wird jedes zweite eingelesen.
- Nach dem fünfzigsten Bild wird jedes dritte eingelesen.
- Nach 100 Bilder wird die Suche eingestellt.
- Sind keine Bilder mehr vorhanden - wird die Suche eingestellt.

Zum Schluss werden alle nicht eindeutig erkannte sowie dem Hintergrund angehörige Pixel auf 0 gesetzt (Zeilen 42-45).

```
01    ImageProcessor maskProcessor=new ij.process.ByteProcessor(width, height);
02    maskProcessor.setColor(0);
03    maskProcessor.fill();
04    findEllipse(imgOriginal, houchModus);
05    maskProcessor.setColor(255);
06    maskProcessor.fillOval(X-A, Y-B, (A)*2, (B)*2);
07    int yStart = (Y-B-1)<0?0:Y-B-1;
08    int yEnd = ((Y-B+(B*2)+1)>height?height:Y-B+(B*2)+1);
09    int xStart = (X-A-1)<0?0:(X-A-1);
10    int xEnd = ((X-A+(A*2)+1)>width?width:(X-A+(A*2)+1));
11    do {
12       tempProzessor =temp.getProcessor().convertToByte(false);
13       pixelFuerDreieck=0;
14       thresholds=fillThresholds(tempProzessor,thresholdLevel, thresholdArt);
15       for (int y=yStart; y<yEnd;y++)
16          for (int x=xStart; x<xEnd; x++)
17             if (maskProcessor.get(x,y)==255 || maskProcessor.get(x,y)==10){
18                prozent=m.calculateProzent(tempProzessor,
19                      getThreshold(tempProzessor, thresholds,x,y), x, y);
20                if       (prozent>=matchRateMax) maskProcessor.set(x,y,1);
21                else {
22                      pixelFuerDreieck++;
23                      if (prozent<=matchRateMin) maskProcessor.set(x,y,10);
24                }
25             }
26
27       weiter=false;
28       for (int y=yStart; y<yEnd;y++)
29          for (int x=xStart; x<xEnd; x++)
30             if (maskProcessor.get(x,y)==255) weiter=true;
31       if (pixelFuerDreieck>(Flaeche/100)) weiter=true;
32       if (weiter){
33          if (frameNumber-startfr>10) frameNumber++;
34          if (frameNumber-startfr>50) frameNumber++;
35          if (frameNumber-startfr<=100){
36             temp=imgLoader.openImage(Dir+imgName+form.format(++frameNumber)+imgFmt);
37             if (temp==null) weiter=false;
38          }else weiter=false;
39       }
40    }while (weiter);
41
42    for (int y=yStart; y<yEnd;y++)
43       for (int x=xStart; x<xEnd; x++)
44          if (maskProcessor.get(x,y)==255 || maskProcessor.get(x,y)==10)
45             maskProcessor.set(x,y,0);
```

Listing 3.13: createMask

3.4 Funktionalität der graphischen Oberfläche

Die graphische Oberfläche bietet einige Einstellmöglichkeiten für den Benutzer, auf die hier kurz eingegangen werden soll. Wie in Abbildung 3.14 zu sehen ist, kann durch die Checkbox (1) bezweckt werden, dass die Maske beim Image Mosaicing automatisch erstellt wird. Der Button (2)

startet die Erstellung der Maske zu Testzwecken. Mit Slider (3) kann man die Prozentzahl vorgeben, ab welcher es zu Nutzbereich oder als zu Hintergrund zu zählen ist. Der gelb markierte Bereich des Sliders bildet den Prozentzahlbereich, in dem das Pixel als nicht eindeutig identifizierbar anzusehen ist. Slider (4) realisiert die Anzahl der Bereiche für lokales Schwellwertverfahren. Per default ist 8 eingestellt. Radiobuttonset (5) definiert das Schwellwertverfahren, das von der Maskenerstellung verwendet werden soll. Die Variante ImageJ verwendet das ImageJ-eigene Verfahren.
Radiobuttonset (6) bestimmt den Filtertyp. 1-bedeutet dabei, dass es nicht mit Filter gearbeitet wird, sondern jedes Pixel ohne Berücksichtigung von Nachbarn bewertet wird. *3,5* und *7* bedeuten Filter 3x3, 5x5 und 7x7 mit Wert 1 in jeder Zelle. *3+* und *5+* dagegen bezeichnen Filter, die in 3.1 und 3.2 abgebildet sind.

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

3.1: 3+

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1.5 & 2 & 1.5 & 1 \\ 1 & 2 & 3 & 2 & 1 \\ 1 & 1.5 & 2 & 1.5 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

3.2: 5+

Radiobuttonset (7) gibt an, welcher Debuglevel benutzt werden soll. Debuglevels wurden im Kapitel 3.2 genau beschrieben. Und schließlich Rodiobuttonset (8) legt fest, welcher Modus bei Houghtransformation gebraucht werden soll.

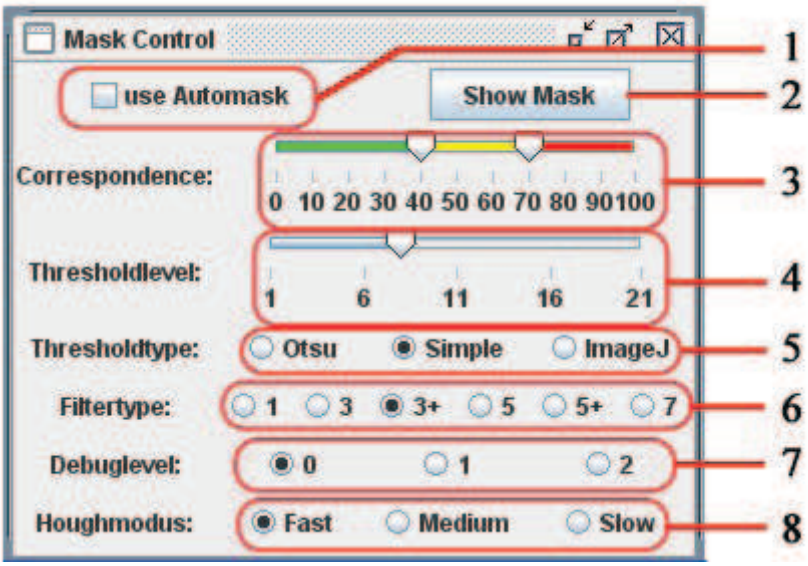

Abbildung 3.14: Graphische Benutzeroberfläche

Kapitel 4

Funktionstest

Eine Zusammenfassung der Erkenntnisse aus vielen Tests soll im Folgenden Aufschluss darüber geben, wie erfolgreich und mit welchen Einstellungen die Maskenerzeugung funktioniert.

Angefangen bei Houghmodus kann man sagen, dass die Version *"Fast" zu* 95% der Fälle erfolgversprechend ist. Dabei ist sie die schnellste, sodass es sich lohnt, Modus *"Fast"* als Defaulteinstellung zu fixieren. In 99% der Fälle liefert Version *"Medium"* gute Resultate. Dabei ist sie nur ein wenig langsamer als *"Fast"*. *"Slow"* verspricht Erfolg zu 100% der Fälle, allerdings dadurch, dass sie außerordentlich rechenaufwändig ist, ist sie in der Praxis kaum einsetzbar.

Filtertyp spielt keine große Rolle. Abweichungen bei verschiedenen Filtertypen sind nur geringfügig. "*3+*" ist auf jeden Fall eine gute Option.

Schwellwertverfahren *"Simple"* hat sich als bestes erwiesen. Es konnte bessere Resultate in 100% der Fälle liefern. "*ImageJ*" berechnet ungefähr denselben Schwellwert wie "*Otsu*". Die beiden versagen oft, wenn es um die Dreiecksuche geht. Allerdings bei der Kantendetektion liefert "Otsu" gute Resultate. Daher ist es fest fixiert, dass nur "*Otsu*" bei der Kantendetektion zum Einsatz kommt.

Thresholdlevel gibt die Anzahl der lokalen Bereichen für Schwellwertverfahren. Daher kann man sagen, dass ein größerer Wert bei Thresholdlevel bessere Genauigkeit bedeutet. Allerdings bei zu großen Einstellung werden die Bereiche so klein, dass das Schwellwertverfahren zu versagen anfängt. In der Praxis hat sich die Anzahl 16 als optimal erwiesen. Ausmaße vom Bild spielen dabei auch eine Rolle und müssen bei der Wahl mitberücksichtigt werden.

Einstellungen des Sliders Correspondens spielen auch keine große Rolle. Pauschal kann man aber sagen, dass die Erhöhung der rechten Regler die Anzahl falsch erkannten Pixel im Nutzbereich verringert (Pixel, die eigentlich zu Hintergrund gehören) und gleichzeitig die Anzahl der falsch erkannten Pixel im Hintergrund erhöht (Pixel, die eigentlich zum Nutzbereich gehören).

Im Allgemeinen kann man sagen, dass die Maskenerzeugung ganz gut funktioniert und anständige Resultate liefert.

Kapitel 5

Fazit

Im Zuge dieser Arbeit konnte die gestellte Aufgabe vollständig realisiert werden. Obwohl die Performance nicht im Vordergrund stand, hat sich während der Implementierung herausgestellt, dass die realisierbaren Algorithmen sehr rechenaufwändig sind. So mussten einige optimierende Schritte unternommen werden um dem Benutzer die Möglichkeit gegeben durch die Benutzeroberflächeneinstellungen auf die Wartezeiten einwirken zu können.
Mit dieser Arbeit wurde ein weiterer Schritt für den Einsatz dieser Software bei medizinischen Operationen geschaffen.
Um die Einarbeitung für zukünftige weitere Arbeiten, die dieses Projekt erweitern werden, zu erleichtern, sind die wichtigsten Knoten der Maskenerkennung, anhand der jeweiligen Codelistings im Text, ausführlich erklärt worden. Der umfassenden Dokumentation wurde hier besondere Aufmerksamkeit erwiesen.

Literaturverzeichnis

- [1] M Kourogi, T Kurata J, et al.: Real-time image mosaicing from a video sequence. Procs ICIP99, vol. 4 133-137, 1999
- [2] Martin Naderi. Bachlorarbeit. Implementierung eines Echtzeitverfahrens zur Erstellung von Bildmosaiken aus endoskopischen Videosequenzen. 2007
- [3] Christian Zimmermann, Diplomarbeit. Implementierung einer Benutzerschnittstelle zur medizinischen Evaluierung eines Image Mosaicing-Verfahrens in der Endoskopie. 2008
- [4] W. Konen, M. Naderi, M. Scholz Endoscopic image mosaics for real-time color video sequences. 2007
- [5] W. Konen, Technischer Report. Optischer Fluss und Echtzeit-Videobearbeitung. 2006.
- [6] D. Lammers, S. Wachenfeld, C. Lordemann, M. Lambers - Seminar: „Unterstützung von Landminendetektion durch Bildauswertungsverfahren und Robotereinsatz“ 2004
- [7] T Borghoff, N. Höll Vortrag Segmentierung. 2005
- [8] Wikipedia.de - Canny-Algorithmus. 2009
- [9] S. Erhard, Studienarbeit - Die Hough-Transformation als interaktive Lerneinheit. 2006
- [10] Wikipedia.de - Hough-Transformation. 2009
- [11] Wikipedia.ru - Преобразование Хафа 2009
- [12] Wikipedia.de - Schwellwertverfahren, 2009
- [13] Wikipedia.ru - Метод Оцу, 2009

Abbildungsverzeichnis

Listingsverzeichnis